A STUDY IN AUTOMATIC PROGRAMMING

CARNEGIE-MELLON UNIVERSITY

PREPARED FOR
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY
AIR FORCE OFFICE OF SCIENTIFIC RESEARCH

MAY 1974

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR - TR- 74 - 1426	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD 784 816
4. TITLE (and Subtitle) A STUDY IN AUTOMATIC PROGRAMMING		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jack R. Buchanan		8. CONTRACT OR GRANT NUMBER(s) F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Department of Computer Science Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101D A02466
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, VA 22209		12. REPORT DATE May, 1974
		13. NUMBER OF PAGES 155
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Air Force Office of Scientific Research /NM 1400 Wilson Blvd Arlington, VA 22209		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A description of methods and an implementation of a system for automatic generation of programs is given. The problems of writing programs for numerical computation, symbol manipulation, robot control and everyday planning have been studied and some programs generated. A particular formalism, i.e. a FRAME, has been developed to define the programming environment and permit the statement of a problem. A frame, F, is formulated within the Logic of Programs (Hoare 1969, Hoare and Wirth 1972) and includes primitive functions

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

I SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

20. (abstract cont.)

and procedures, axioms, definitions and rules of program composition. Given a frame, F, a problem for program construction may be stated as a pair $\langle I, G, \rangle$, where I is an input assertion and G is an output assertion. The program generation task is to construct a program A such that $I\{A\}I'$, where $I' \supset G$. This process may be viewed as a search in the Logic of Programs for a proof that the generated program satisfies the given input-output assertions. Correctness of programs generated using the formal algorithm is discussed.

A frame is translated into a backtrack problem solver augmented by special search procedures. The system is interactive, responds to simple advice and allows incremental and structured program development.

The output or solution program is a subset of ALGOL containing procedure calls, assignments, while loops and conditional statements.

II

MAY 1974

A STUDY IN AUTOMATIC PROGRAMMING

By

Jack R. Buchanan

ABSTRACT

A description of methods and an implementation of a system for automatic generation of programs is given. The problems of writing programs for numerical computation, symbol manipulation, robot control and everyday planning have been studied and some programs generated. A particular formalism, i.e. a FRAME, has been developed to define the programming environment and permit the statement of a problem. A frame, F, is formulated within the Logic of Programs [Hoare 1969, Hoare and Wirth 1972] and includes primitive functions and procedures, axioms, definitions and rules of program composition. Given a frame, F, a problem for program constuction may be stated as a pair <I,G>, where I is an input assertion and G is an output assertion. The program generation task is to construct a program A such that I{A}I', where I' ⊃ G. This process may be viewed as a search in the Logic of Programs for a proof that the generated program satisfies the given input-output assertions. Correctness of programs generated using the formal algorithm is discussed.

A frame is translated into a backtrack problem solver augmented by special search procedures. The system is interactive, responds to simple advice and allows incremental and structured program development.

The output or solution program is a subset of ALGOL containing procedure calls, assignments, while loops and conditional statements.

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contracts [F44620-73-C-0074] and [DAHC15-73-C-0435].

ACKNOWLEDGEMENTS

I am very grateful to my advisor Dr. David C. Luckham for his constant encouragement and guidance throughout the course of this research. He has contributed many ideas and refinements to this work as it has progressed.

Many others have contributed to the development of this research through discussion and programming suggestions to which I am very grateful. These include Bruce Baumgart, Tom Binford, Horace Enea, Richard Fikes, Peter Hart, John McCarthy, Nils Nilsson, Richard Orban, David Smith, Dan Swinehart, and Larry Tesler.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

During the 1950's the phrase "automatic programming" described the process carried out by assemblers and compilers, i.e. the translation of a program written in one language into another where the "meaning" is preserved and the target language is interpretable. Since then there have been many advances in programming languages and their associated processors allowing the user to specify at higher levels, or in more natural ways, how the computation should proceed and removing the users responsibility for such things as storage management, resource allocation, etc.

In the present project, we have sought to develop methods that will further automate or augment the programming process by generating programs over several domains in a suitably defined environment given a statement of what they are to accomplish, i.e. programming by assertion rather than algorithmically. We seek to generate programs using a statement of the desired program's properties rather than compiling from one detail specification of the flow of control into another. It is in this sense that we use the term automatic programming. Automatic programming may be further distinguished from compiling by its use of a semantic model together with a deduction capability. It is to be expected, however, that as progress is made in automatic program generation that research in compilation will be benefited.

As the field of Artificial Intelligence has matured, problem solving techniques have been developed that have allowed us to seriously consider building automatic programming system. Some very influential ideas came from the Heuristic Compiler [Simon 1963] and the GPS [Newell and Simon 1963] projects, i.e. the notion of building up a program in a state-space tree search using a problem reduction procedure. This is certainly basic, almost subconsciously so, to the present project and has been widely used by others.

INTRODUCTION

During the 1960's much of the theory of problem solving was associated with tree or graph searching methods. Well known techniques for restricting the search by using evaluation functions, "minimaxing", the α-β method, etc. may be found in [Nilsson 1971], [Samuel 1963]. Later automatic programming work, still depending heavily on search strategies, sought to represent the domain semantics and carry out the deduction in first order logic using the principle of resolution [Green 1969], [Waldinger and Lee 1969]. A more powerful (generated deeper proofs) general deduction system combining resolution, equality and algebraic simplification was reported in [Allen and Luckham 1970]. A great deal of the systems' efforts were spent in search because most facts were uniformly represented as axioms in clause form and the search strategies were largely syntactic. Greater efficiency was gained in a system built by separating the heuristic search from the deduction and employing the GPS paradigm [Fikes and Nilsson 1971].

The difficulty in these systems of using facts to guide the search has prevented them from solving hard (for humans) problems or generating complex programs. It has become clear that in addition to a manageable basic problem solving method, knowledge, both general and domain specific, must be provided in a functionally useful way to enable the system to find a solution. During the last two years language systems that allow the user to easily embed knowledge at all levels have been developed. Other useful features are pattern matching, pattern evoked procedures, flexible control structures, multiple contexts and processes [Hewitt 1969], [Sussman and McDermott 1972], [Rulifson et al. 1972], [Feldman et al. 1972], [Tesler, Enea and Smith 1973]. Our use of some of these features will be described in later sections. Taking advantage of some of these features and refining the notion of semantics was a

natural language understanding system reported in [Winograd 1971]. Other automatic programming or debugging systems are [Deutsch 1973] and [Sussman 1973]. Some general descriptions and useful classifications of the components of an automatic programming system have been given in [Balzer 1972]. The closure of all these capabilities is yet to be fully exploited in a problem solving or automatic programming system.

Procedural knowledge may be distinguished from declarative in that the information content is expressed within the flow of control of a computation (in the general sense) sequence, i.e. the data from which useful information may be extracted is the program itself. This is probably the most efficient information access scheme of all. An intelligent system in which all information is expressed procedurally will rely more heavily (perhaps totally) on the current state of computation to determine its future behaviour than a system utilizing declarative facts and also be, necessarily, more dependent on the ordering of its knowledge.

However, the distinction begins to blur when we consider how a system may effectively utilize declarative information and how given a general computational model, e.g. problem reduction algorithm as in our system, declarative facts may be translated into procedures. Another example of this is the "questionnaire programming" approach for customizing business application systems. Progress has been made in defining model specification languages having procedural and non-procedural components [Hewitt 1969],[Martin 1973], [Hammer, Howe and Wladawsky 1974]. Even a resolution based theorem prover with an appropriate protocol language (the procedural part) can efficiently use its knowledge to solve a problem [Allen and Luckham 1973], [Stickel 1974].

Research in verifying existing programs [Floyd 1967], [King 1969], [Katz and Manna 1973], [Milner 1972] has contributed to our understanding of programs and we have found (not surprisingly) that the kinds of facts required to verify programs are not distinct from those required for the synthesis of correct programs. Progress has also been made in defining axioms and rules of inference for the semantics of programming languages [Hoare 1969] and in particular with respect to the programming language PASCAL [Hoare and Wirth 1972]. This Logic of Programs has been further developed and used as a basis for a verification system in [Igarashi, London and Luckham 1973]. As a logical basis for an automatic programming system this logic is especially convenient since the rules are intuitively clear, the system operation may be easily formalized and correctness considered, and rule applications proceed in natural (for humans) steps.

The objectives of the present project have been to extend the theory of semantic definitions to describe automatic programming problems and to design and implement a system that uses this information in a functionally useful way to atuomatically, or interactively, generate programs.

The particular formalism developed to define the programming environment (or FRAME) called the FRAME language, will be shown to have elements whose form corresponds to statements in the Logic of Programs. It is based on a typed, free variable first order logic in which statements may have truth values of either true, false or undetermined. The frame language consists of primitive procedures, logical axioms, definitions, iterative schemes and additional information about these rules and the relations in them. Other rules of program composition, referred to as standard rules (described in Section 2), are built into the system and needn't be specified for each frame, i.e. composition rule, conditional rule, etc.

The frame language may be viewed as an intermediate level model specification language that is non-procedural and domain independent. It was motivated by observing that in the general deduction systems previously mentioned there was more information in the axioms than was being used operationally, i.e. there were different kinds of axioms and relations (see Section 3) that should be treated differently by the system. For example the truth value of some relations are functions of the state, or FLUENT [McCarthy and Hayes 1969] and some are NON-FLUENT. For efficiency some relations could be handled in a two-valued logic, i.e. TOTAL, and others require the generality of a three-valued logic. Also search guidance information should be provided (embedded) at all levels. For example compared with a resolution based system we would like to choose the best "set-of-support" at each level of deduction. We also wanted a language extendible to, or translatable from a higher level, more natural input language, e.g. recursion equations for the Fibonacci series example in Section 3 and the factorial example in Section 6. A frame actually describes programming techniques, the extensiveness of which determine the complexity of programs produceable using it.

Given a frame, F, a problem for program construction may be stated as a pair <I,G>, where I is an input assertion and G is an output assertion. The program generation task is to construct a program A such that I{A}I', where I'⊃G. This process may be viewed as a search in the logic of programs for a proof that the generated program satisfies the given input-output assertions. A solution to the problem is the sequence of rules of inference and axioms used in the proof. This view allows us to show correctness of the formal methods for program construction. The correctness of the program actually generated by the system will depend on our ability to implement

the formal algorithm. The solution, or output, programs are written in a subset of ALGOL containing procedure calls, assignments, while loops and conditional statements. Program construction is by simulated execution where iterative rules with associated output assertions are used to update the computation model for simulating the execution of a loop.

The application domains studied and in which programs have been generated are numerical computation, symbolic manipulation, guidance procedures for a robot, assembly and repair of machinery, and sequential planning together with generating contingency plans for a wide range of decision making problems. Though we have here pursued a course of developing one system then applying it to several domains by merely changing the content of the frame definitions, it is expected that for practical performance, the form of the system definitions will depend on the domain. For example this is currently happening in research attempting to apply this system to automating data base management tasks [Gerritsen 1974] and automated repair of machinery [Luckham and Buchanan 1974].

The rules of inference, axioms and other logical facts expressed in the frame definitions are translated into a backtrack problem reduction system augmented by special search procedures using these facts. The target language of this translation is LISP using primitives and backtracking facilities of Micro-Planner [Hewitt 1971], [Sussman and Winograd 1972]. This subgoaling system recursively applies to a goal the rules of the frame to generate subgoals whose solution imply a solution to the original goal.

As an auxilary to the subgoaling system is an ADVICE system with an associated language that allows the user to guide the search, modify the frame, restrict rule

applications and receive interactive feed-back during program construction. This is described in Section 3.

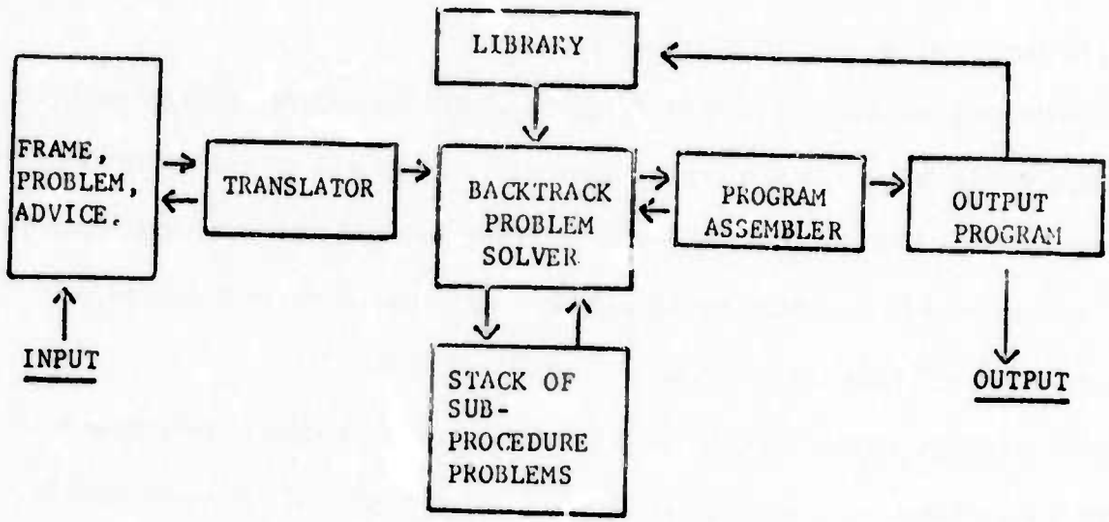


Figure 1. Main System Components

The main components of the system are shown in figure 1. The user may interactively specify a frame and provide some initial advice (model acquisition phase). This is eventually translated into a subgoaling problem solver to which a problem may be given, i.e. a goal which the problem solver seeks to achieve using the rules of the frame (program generation phase). If a solution program is constructed, the user may incrementally extend it, i.e. pose another problem which takes the output assertion of the current solution program as its input assertion. The user may also optimize it, or generalize it and place it in the program library for future inclusion in a generated program. If the program contains conditional calls to as yet ungenerated procedures (see Section 5), these subproblems may be attempted. Subproblems may also arise by declaring some primitive procedures defined in the frame to be assumptions to be expanded into concrete programs. This provides a rather rudimentary, at this time, interactive structured program development facility.

1.1 CONTRIBUTIONS

Some of the areas of work along which progress has been made and contributions to the field may be noted are as follows:

(1) Extending the theory of semantic definitions for defining semantics of programming languages to define automatic program generation environments. A relation has also become more clear betwee: the kind of assertions needed to verify programs and those required to synthesize correct programs, e.g. compare loop invariants used in our system with inductive assertions for program verification.

(2) A prototype system has been developed that is useful in a study to determine the feasibility of building an automatic programming system to augment the programmer in the following ways:

(a) Automatic or interactive generation of possible solution programs for application domains suitably described,

(b) The usefulness of an automated system to handle bookkeeping details,check consistency, applicability, etc.,

(c) The feasability of an interactive structured development system,

(d) The feasability of interactively building up complex programs by allowing incremental program extension, library access, structured development, and contingency planning.

(3) A demonstration is made that declarative facts can be incorporated(translated) into an efficient problem solving search procedure which uses these facts at all levels of search.

(4) A typed, free variable first-order logic in which statements may have truth values of true, false, or undetermined has been shown to be a natural logical basis for automatically generating conditional statements in a program.

(5) The iterative rule computation scheme has a correspondence to the principal of mathematical induction and is a useful way to represent loop structure for a program to be generated.

To the nagging question that it may be as hard (or harder) to specify a frame as it is to write the program, the following answers may be given:

(1) Yes, but we are learning how to program by assertion, and develop defining formalisms and methods for efficiently manipulating facts and rules.

(2) A frame may contain many atomic units of information whose interaction when faced with a novel goal is not easily predictable. For example the robotics frame defined in Section 7 may be used to generate many different programs.

(3) An interactive facility for constructing programs with the extendable features mentioned above can potentially augment the human programmer.

(4) Experience with our frame language has been helpful in investigating the basic information required to construct programs, now the task of raising the level of language interaction to a more natural (and useful) level will be aided.

1.2 EXTENSIONS

The following specific research problems are suggested as natural extensions of this work (i.e. problems we didn't solve):

(the reader may want to scan these now then come back to them after reading further)

(1) In the area of conditional statement generation:

(a) Introduce probabilistic decision theory to determine preference among contingency problems.

(b) Develop criteria for recognizing equivalent or similar subproblems.

(c) Design a more flexible mechanism for managing scope, program structure and contingency goal selection. Since there is no reason to prefer the trunk path, the structure of the output program should not be fixed from that point on.

(d) Compute completely correct input-output assertions for programs having arbitrary nesting of conditional statements.

(2) In the area of automating structured programming:

(a) Develop a human engineered interactive system. Regardless of how the "theology" says we should program, there is something basic to the human condition about how we do program and style improvement must be made within that framework.

(b) Develop techniques for managing side effects.

(c) Do lookahead or design a bottom-up, outside-in, etc. component.

(3) In the area of generating programs with looping structure:

(a) Implement some form of the recursion rule[Hoare 1969].

(b) Develop efficient and more complete methods for updating the state consistently. Design criteria for detecting inconsistent states and prevent them from invalidating the program.

(c) Generate while loops but reduce the information that the user must provide. For example, in iterative rules the system should reasonably deduce the control test or output assertion.

(d) Build in the iterative rule (analogous to the way the conditional rule is built in). This is really trying to do induction. We would like the ability to analyze a computation trace, recognize loop structure and generate a while loop.

(4) A higher level or more comprehensive input language should be developed. It will probably be domain dependent.

(5) Explore the implications of various logics for programs as a basis for automatic programming. In [McCarthy and Hayes 1969] various logics are discussed for intelligent systems.

(6) Strive to free the problem solver from being so dependent on the ordering of goals in a condition to be achieved or the ordering of applicable rules. Develop reordering strategies, lookahead, etc.

(7) In the area of parallel processes:

(a) Generate programs for parallel machines.

(b) Develop criteria for splitting up a generated sequential program into subtasks for cooperating sequential processes.

(8) Exploit multiple processes and multiple contexts to increase the power of the problem solvers, e.g. a better answer to the question of why a node failed could yield automatic correction.

(9) Organize a library of generated programs and develop strategies for its access.

(10) Study the problem of validation of program specification. Determine consistency and adequacy of a programming model. Prove properties of the family of programs constructable from the same frame. Study the invariants of data structure under application of a family of programs, e.g. do they modify the tree orderedness of a label table.

1.3 COMMENTS ON THE FUTURE OF AUTOMATIC PROGRAMMING

The need for some automation in the task of software production is becoming increasingly clear. System are getting bigger and more complex which has caused maintenance cost to rise (It is now 50 per cent of the programming budget). Software costs too much, it isn't reliable, takes too long to develop and its difficult to modify or fix. Programming has not attained the maturity to develop standard engineering practices with their attendant reliability that other disciplines have. Research in automatic programming seeks to understand the nature of the task and thereby improve production.

There are many dimensions along which automatic programming will progress. There is the theoretical dimension which implies gaining a more fundamental understanding of the meaning of programs and developing descriptive and useful logics for automatic programming problems that permit a rigorous investigation of the properties of a program. Along the pragmatic dimension, we will be interested in augmenting current practice with state of the art techniques. There is also the heuristic dimension which contains the multitude of ideas, systems and ad hoc notions for which there is no good logical description nor is there any current practical application, but through them we gain understanding into the nature of the problem. The following are a list of rather random comments on the future of automatic programming based on our experience.

(1) More emphasis will be placed on higher level descriptive formalisms and programming languages to define programming environments. The level will be raised to accomodate the non-programmer as well as to ease the job of the professional. Some of these advances will require major breakthroughs in Artificial Inteliigence, e.g.

dynamic acquisition of models, recognition of incomplete or inconsistent models, or further development in representing knowlege in a functionally useful way.

(2) Larger software facilities will be developed for systems to contain more facts. Deduction will be efficiently encoded (perhaps specialized as in the theorem prover over the integers in [King and Floyd 1970]).

(3) Specialized domain application systems will be built that will rival human abilities (perhaps the standard five year time estimate will do). Compared with the present system these will require new kinds of built in facts, different advice needs and computation schemes. To make real progress transferring technology developed in one system to the improvement of another in perhaps a different domain we must focus on the methods used to embed knowledge or define the environment rather than just loading the system with facts and ad hoc tricks or using a human interface that only its creator can understand. The field is so young that too much time shouldn't be spent hand tuning a system once the basic methods are exploited.

(4) There are some short term payoffs (within 5 years) for augmenting programmers, e.g. better interactive debugging systems, languages permitting user assertions to be checked and better optimizers. Within a narrow domain present technology can yield good performance. Automatic programming will not replace the programmer but will raise the educational level for those who would do computer assisted program construction. With respect to program synthesis we should strive to generate programs of the type that people understand and can write with some effort so that program synthesis does not get completely lost in futuristic AI research. Within current technology the size of the generateable programs will be small (one page) and complexity will be gained by combining and extending them with interactive aids.

(5) INTERACTIVE systems will be developed that will do mundane logical checking, answering "what if" questions, and building up complex programs modularly such that the system will only have to focus on one small problem at a time.

(6) Within the forseeable future final production level systems will not be automatically produced but the ability to produce prototype systems quickly to test design ideas will be a significant aid to software production.

In Section 2 a short description of the logic of programs is given in which the frame definitions and program construction rules are formulated. A simple example is given that illustrates how a problem is formulated and the meaning of a solution. Section 3 describes the frame definition language, advice language and output program language. In Section 4 the systems use of information during the problem solving process is described. Sections 5 and 6 present the system methods for generating conditional statements and iterative loops respectively. Section 7 descibes the programming aids provided in the system for the user to interactively generate more complex programs. In Section 8 is given the formal program generation algorithm and a description of the proof of its correctness. Section 9 is intended to document the system implementation to the level that would be reasonably useful in designing an expanded system. Illustrative examples of frames and generated programs are given in Sections 3,5, 6,7 and Appendix A. Appendix B contains a complete interactive session.

2. LOGICAL BASIS FOR SEMANTIC DEFINITIONS

In this section we will briefly describe how frames can be formulated within the Logic of Programs. Later sections will expand on the frame formalism and its use. Program generation may then be viewed as a search for a proof within the Logic of Programs that the generated program satisfies its input-output assertions. In Section 8 the formal algorithm will be given and correctness of solutions considered.

A distinction should be made between the problem solving algorithms and their implementation in any particular system where an implemented system must fall short of the formal algorithm. For example program correctness will depend upon maintaining consistency of each state occuring during program construction, yet in general the task of determining state consistency is undecidable. However limited deduction is carried out and special mechanisms to detect common inconsistencies, e.g. single valuedness of program variables, are implemented.

NOTATION:

x,y,z,u,v,w...variables,
X,Y,Z,... lists of variables,
f,g,h.... functions,
s,t... functional terms,

G,I,P,Q,R,S,... Boolean expressions (essentially formulas of first order logic with standard functions and predicates for equality, numbers, lists and other data types),

P(X) denotes the formula obtained by replacing each free variable in P by a new variable from X,

$(\exists X)P(X)$ denotes existential quantification over all X-variables in P(X),

A,B,C,... programs and program parts in an Algol-like plan language (details in Section 3),

p,q,... procedure names,

$\alpha,\beta,\lambda,...$ substitutions of terms for variables, also denoted by $(\langle x \leftarrow t \rangle)$.

P(t) denotes the result of replacing x by t everywhere in P(x).

$\alpha\beta$ denotes the COMPOSITION of α and β; $E\alpha\beta = (E\alpha)\beta$ for all expressions E.

We assume the existence of a fixed arbitrary ordering of literals defined in the frame (atoms and negations of atoms) which is simply used as a computational aid for describing and implementing the rule of invariance defined in Section 2.2 and not for any heuristic advantage.

2.1 LOGIC OF PROGRAMS

We review briefly the elements of an inference system for proving properties of programs [Hoare 1969]. This description is taken from [Igarashi, London, Luckham 1973].

STATEMENTS of the logic are of three kinds:

(i) Boolean expressions, (henceforth often called ASSERTIONS)

(ii) statements of the form P{A}Q where P,Q are Boolean expressions and A is a program or program part.

P{A}Q means "if P is true of the input state and A halts (or halts normally in the case that A contains a GO TO to a label not in A) then Q is true of the output state".

(iii) Procedure declarations, p PROC K where p is a procedure name and K is a program (the body of p).

A RULE OF INFERENCE is a transformation rule from the conjunction of a set of statements (premisses, say H_1 ,...,H_n) to a statement (conclusion, say K) of kind (ii). Such rules are denoted by

$$\frac{H_1,\ldots,H_n}{K}$$

The concept of PROOF in the logic of programs is defined in the usual way as a sequence of statements that are either axioms or obtained from previous members of the sequence by a rule. A proof sequence is a proof of its end statement.

NOTATION: We use H ||- K to denote that K can be proved by assuming H. H |- K denotes the same thing for first order logic. It is sometimes helpful to denote statements that are problems or subproblems for the program generator to solve by P{?}Q.

2.2 FRAME RULES

The RULES in a frame F are of three kinds:

(a) PROCEDURES transform states into states and are expressed as statements in the logic of programs.

(b) SCHEMES are methods for constructing programs and are expresed as rules of inference in the logic of programs.

(c) RELATIONAL LAWS: definitions and axioms which hold in all states and serve to "complete" incomplete state descriptions by permitting first order deduction of other elements of a state from those given.

Given a frame F a problem for program construction may be stated as a pair <I,G>, where I is an input assertion (or initial state) and G is the output assertion (or goal that must be true in the output state). The program construction task is to construct a program A such that I{A}I', where I'⊃G. A solution is the sequence of rules of F used in the construction of the solution program A.

NOTATION and RESTRICTIONS: $Q \cup F \supset R$ denotes that R is a logical consequence of Q and the axioms of F. Assertions describing states are denoted by I,I',...,G,G',... These assertions (but not the assertions in rule definitions) are restricted to be conjunctions of atomic assertions. We write R∈I to denote that R is a conjunct in I. L(F) denotes the logic of F,i.e. the set of consequences of the rules of F. Substitutions $\propto$ do not replace any variable that occurs in the initial state I. Expressions, all of whose variables occur in the initial state are called "fully instantiated".

STANDARD FRAME RULES: A set of standard rules are assumed to be part of every frame. These are rules implemented in the program construction methods of the problem solving algorithm:

R0. Assignment Axioms:

(i) Simple Assignment: $P(t)\{x \leftarrow t\}P(x)$

(ii) Conditional Assignment: $(\exists Z)P(Z)\{\text{IF } P(W) \text{ THEN } Y \leftarrow W\}P(Y)$
$\neg(\exists Z)P(Z) \wedge Q(Y)\{\text{IF } P(W) \text{ THEN } Y \leftarrow W\}Q(Y)$

where Y-variables in P(Y) do not occur in P(W), W-variables are special variables ocurring only in conditional assignments, and Y←W denotes the sequence of simple assignments between members of Y and W that occur in the same argument positions in P(Y) and P(W).

R1. Rule of Consequence: $\dfrac{P \supset Q, Q\{A\}R \quad P\{A\}Q, Q \supset R}{P\{A\}R \qquad P\{A\}R}$

R2. Rule of Composition: $\dfrac{P\{A\}Q, Q\{B\}R}{P\{A;B\}R}$

R3. Rule of Invariance: if $P\{A\}Q$ and $I \cup F \supset P$ then $I\{A\}Inv(Q,I)$
where if $R_1, R_2, \ldots, R_n$ are the conjuncts of I
in the fixed order, then $I_0 = Q$,
for $0 \leq m < n$, $I_{m+1} = I_m \wedge R_m$ if $\neg(I_m \cup F \supset \neg R_m)$
$I_{m+1} = I_m$ otherwise,
and $Inv(Q,I) = I_n$.

R4. Change of Variables: $\dfrac{P(x)\{A(x)\}Q(x)}{P(y)\{A(y)\}Q(y)}$ where y is not a special variable.

R5. Conditional Rule: $\dfrac{P \wedge Q\{A\}R, \quad P \wedge \neg Q\{B\}R}{P\{\text{IF } Q \text{ THEN } A \text{ ELSE } B\}R}$

R6. Undetermined values: If $I'\{?\}G$ cannot be solved and $\neg(I' \cup F \supset \neg G)$ then G is UNDETERMINED in I'.

STANDARD RULES

REMARKS: (i) The axioms R0(ii) define the semantics of conditional assignment statements used primarily in the system during the assembly of while loops. The

relation P(W) within the IF statement is interpreted as a call to a Boolean procedure that, if successful, will bind the W-parameters to values from the state that make it true. Our convention is to regard W-variables as "special variables" only occurring in such conditional assignments. An alternative would be to define a typed procedure for each relation in the frame that would return the appropriate value for direct assignment to the Y-variables. We felt that the conditional assignment made the desired semantics more transparent however.

(ii) The rule of invariance means that during a state transformation and a new statement Q becomes true in I that the function Inv(Q,I) will return Q union these facts in I that do not contradict Q. We therefore do not need "frame axioms" to handle the "frame problem" [McCarthy and Hayes 1969] as the resolution theorem provers mentioned in the introduction did.

(iii) The rule of undetermined values guides the systems decision to generate conditional statements (Section 5).

INPUT FRAME RULES: In addition to the standard rules, a frame may contain rules of the following types (these constitute the user defined elements of the frame):

S1. Primitive procedures (or operators): the rule defining procedure p is of the form P{p}Q. The assertions P and Q are the pre- and post-conditions of p. p must contain a procedure name and parameter list.

S2. Iterative rules: an iterative rule definition containing the Boolean expressions P(basis), Q(loop invariant), R(iteration step goal), L(control test) and G(rule goal) is a rule of inference of the form:

(a) $\dfrac{P \vdash Q,\ Q \wedge L\{?\}R,\ R\{??\}Q \vee \neg L}{P\{\text{while } L \text{ do } ?;??\}G}$

where the free variables of R and L occur in Q. Such rules are permitted not to contain P or L,in which case they correspond to inferences of the form:

(b) $\dfrac{Q,\ Q \wedge \neg G\{?\}R,\ R\{??\}Q \vee G}{Q\{\text{while } \neg G \text{ do } ?;??\}G}$

S3. Definitions. A definition of G in terms of P is a logical equivalence $\vdash P \equiv G$.

S4. Axioms. A frame axiom P is a logical axiom $\vdash P$.

Terms and predicates in assertions may contain calls to LISP functions. If the frame definition contains functional terms or predicate tests that are evaluated by calls to LISP functions, the set of premisses must be expanded to include both the input-output assertions for these function calls and the logical axioms for the relevant data types.

REMARKS (i) The iterative schemes S2 permit the definition of methods for constructing loops; they are instances of:

WEAK ITERATION RULE: $\dfrac{Q \wedge L\{B\}Q \vee \neg L}{Q\{\text{WHILE L DO B}\}\neg L}$

where Q is the invariant of the loop. The meaning of $\vdash Q$ in the premiss is that the rule may only be applied in states where Q is a first order consequence of the state description. The program part ?? is restricted to be a sequence of assignment statements (see Section 6).

(ii) Inconsistencies may arise in several different ways in frames. The axioms can be inconsistent, or the post conditions of a rule can be inconsistent with the axioms. Also the elements of iterative schemes must satisfy some simple consistency criteria (section 6).

(iii) Note that each frame rule has a goal. The goal of a procedure is its postcondition; the goal of an axiom or definition is its consequent.

The following lemma is useful in proving properties of conditional assignments [Igarashi,London,Luckham 1973]:

OR-LEMMA $\dfrac{P\{A\}Q,\ R\{A\}S}{P \vee R\{A\}Q \vee S}$

2.3 A SIMPLE ROBOTIC EXAMPLE

We will now consider a simple robotics environment and its description within the formalism. In the context of this example we will then consider formulating the correctness of solutions.

Consider the following frame and problem:

INPUT FRAME RULES:

F1. Procedure: standon

AT(x,y)∧AT(z,y)∧ROBOT(x)∧BOX(z){standon(x,z)}ON(x,z).

F2. Procedure: step-up

ROBOT(x)∧ON(x,y)∧STACKED(z,y){step-up(x,y,z)}ON(x,z).

F3. Iterative Rule: climb

ROBOT(M)∧ON(M,y)∧STACKED(u,y)∧¬ONTOP(M){?}ON(M,u)

ROBOT(M)∧ON(M,y)∧STACKED(u,y){WHILE¬ONTOP(M)DO BEGIN ?;?? END}ONTOP(M)

F4. Axiom: ROBOT(x)∧∃y(ON(x,y)∧∀z¬STACKED(z,y))↔ONTOP(x).

PROBLEM

I: ROBOT(M)∧BOX(B1)∧BOX(B2)∧BOX(B3)∧AT(B1,L)∧AT(M,L)
∧STACKED(B2,B1) ∧ STACKED(B3,B2).

G: ONTOP(M)

PROBLEM 1: CLIMBING

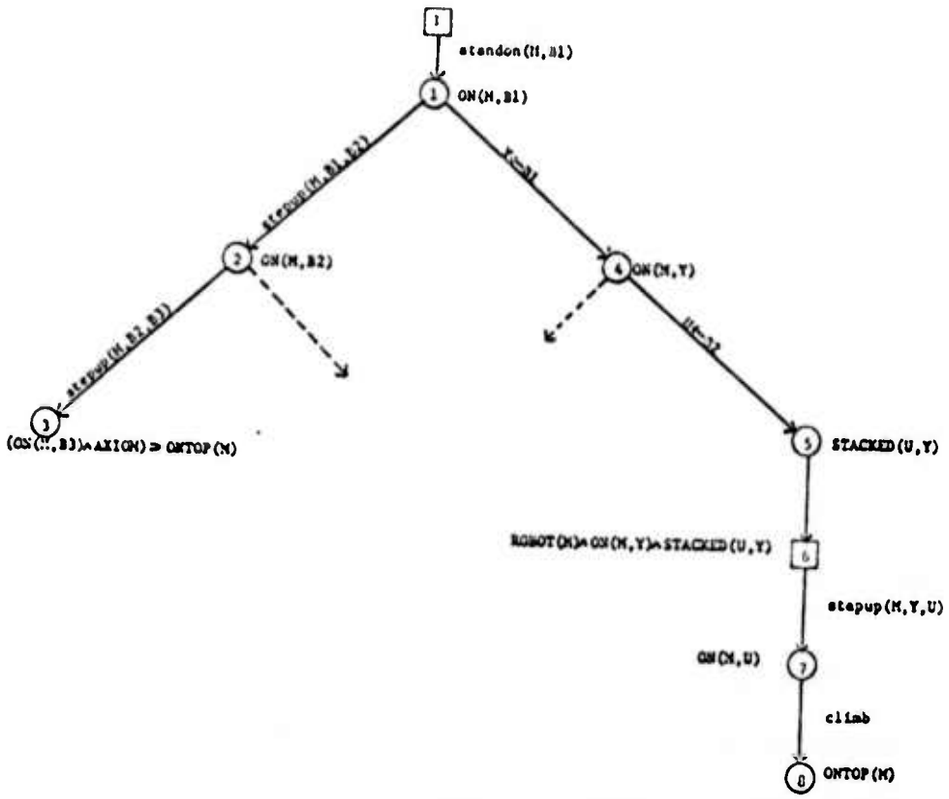
REMARKS: (i) The iterative rule says "A solution to the problem of climbing one box at a time, can be used to construct a WHILE loop that solves the problem of climbing a stack of boxes". The rule defines the meaning of WHILE in the environment. Or we may regard the rule as an induction principle for the environment.

(ii) The program part ?? in the conclusion of the iterative rule transforms the situation after the execution of the loop body (?) back into one in which the invariant is again true if the test is true:

ON(x,u){??}ROBOT(x)∧ON(x,y)∧STACKED(u,y).

We restrict ?? to be a sequence of assignments.

(iii) The goal of climb is ONTOP(M), the negation of the control test in this example.



SEARCH FOR SOLUTIONS TO THE CLIMBING PROBLEM
Figure 2

Steps taken by a search procedure in solving this problem are shown in figure 2. It starts with state situation I and determines by logical reasoning from I and the axioms which operators have pre-conditions that are true in I . It applies these operators and updates the state to the new state using the rule of invariance. It repeats this process on the new states. Node 6 indicates the initiation of a subproblem (the premiss of the iterative rule) with a new initial state (the invariant) which is a subset of the state above it at Node 5. The solutions corresponding to the paths shown in figure 2 are:

(i) I{standon(M,B1);stepup(M,B1,B2);stepup(M,B2,B3)}ONTOP(M).

(ii) I{standon(M,B1);y←B1;u←B2;
WHILE ¬ONTOP(M) DO BEGIN

```
stepup(M,y,u);
y←u;
IF STACKED(w,y)THEN u←w;
END}ONTOP(M)
```

where the assignments within the WHILE loop correspond to the ?? of the iterative rule. The variable w is a special variable.

Using the frame rules we can now construct a proof of the statement I{solution}G within the logic of programs.

1. I⊃(ROBOT(M)∧AT(M,L)∧AT(B1,L)∧BOX(B1))

2. I{standon(M,B1)}ON(M,B1)∧STACKED(B2,B1)∧ROBOT(M) 1,F1,R4,R1,R3

3. ON(M,B1)∧STACKED(B2,B1)∧ROBOT(M){y←B1;
   u←B2}ROBOT(M)∧ON(M,y)∧STACKED(u,y) R0(i),R2,R3

4. I{standon(M,B1);y←B1;u←B2}ROBOT(M)∧ON(M,y)∧STACKED(u,y) 2,3,R2

5. ROBOT(M)∧ON(M,y)∧STACKED(u,y){stepup(M,y,u)}ON(M,u)∧ROBOT(M) F2,R4

6. ROBOT(M)∧ON(M,u){y←u}ROBOT(M)∧ON(M,y) R0,R3

7. ON(M,y)∧∃zSTACKED(z,y){IF STACKED(w,y)THEN u←w}ON(M,y)∧STACKED(u,y) R0,R3

8. ¬∃zSTACKED(z,y)∧ONTOP(M){IF STACKED(w,y)THEN u←w}ONTOP(M) R0

9. (ON(M,y)∧∃zSTACKED(z,y))∨(¬∃zSTACKED(z,y)∧ONTOP(M))
   {IF STACKED(w,y)THEN u←w}(ON(M,y)∧STACKED(u,y))∨ ONTOP(M) OR-Lemma 7,8.

10. ROBOT(M)∧ON(M,y)∧¬(∃z)STACKED(z,y) ⊃ ONTOP(M) F4,
    ⊃(ON(M,y)∧∃zSTACKED(z,y))∨ONTOP(M)
    ROBOT(M)∧ON(M,y)∧ ∃zSTACKED(z,y) ⊃ (ON(M,y)∧∃zSTACKED(z,y))∨ONTOP(M)
    ROBOT(M)∧ON(M,y) ⊃ (ON(M,y)∧∃zSTACKED(z,y))∨ONTOP(M)

11. ROBOT(M)∧ON(M,y)∧STACKED(u,y){stepup(M,y,u);y←u;
    IF STACKED(w,y)THEN u←w}(ON(M,y)∧STACKED(u,y))∨ ONTOP(M) 5,6,10,9,R2,R1

12. ROBOT(M)∧ON(M,y)∧STACKED(u,y){WHILE¬ONTOP(M) DO ...}ONTOP(M) 11,R1,F3

13. I{solution (ii)}ONTOP(M) 4,12,R2

PROOF of I{solution (ii)}G

We refer to a formal proof of L(F)||-I{A}G as a correctness proof. The existence of such a proof implies only that the program is correct relative to the frame. If we modify the frame we can investigate the correctness of solution (ii) in the extended frame by analyzing the proof of I{solution (ii)}ONTOP(M) by checking to see if any step uses facts from an intermediate state situation I' that contradict the extra logical rules. We in effect carry out a "proof checking" operation for consistency of each step with the additional facts. This process practically avoids search.

3. DEFINING THE PROGRAMMING ENVIRONMENT

In this section the Frame definition formalism is presented. This includes the Frame language the Advice language, and the output Program language. A complete example of an input frame, together with advice, and the resulting output program is given.

3.1 FRAME LANGUAGE

3.1.1 ASSERTIONS: The syntax for assertions used in definitions of rules, axioms and state descriptions is shown in figure 3.

```
<variable> ::= <identifier>
<function symbol> ::= <identifier>
<predicate symbol> ::= <identifier>
<term> ::= <variable>|(<function symbol>)|
        (<function symbol><argument list>)
<argument list> ::= <term>|<term>,<argument list>
<functional term> ::= (EV<term>)|(EVN<term>)|<term>
<atomic formula> ::= <predicate symbol>(<predicate argument list>)
<predicate argument list> ::= <functional term>|<functional term>,
                              <predicate argument list>
<literal> ::= <atomic formula>|¬<atomic formula>
<literal element> ::= <literal>|REQUEST(<literal>)|{<assertion>}
<disjunction> ::= <literal element>|<literal element><or><disjunction>
<assertion> ::= <disjunction>|<disjunction><and><assertion>
<and> ::= ∧|&
<or> ::= ∨|⊕
```

SYNTAX OF ASSERTIONS
Figure 3.

Identifiers are strings of characters not containing the negation symbol, "¬", nor the usual LISP delimiters, e.g., blanks, commas or parentheses. The <or> connectives have higher precedence than the <and> connectives and a logical condition is terminated by a semicolon, ";". For example,

$P(x) \vee Q(x) \wedge R(x,y) \wedge S(Z,x) \vee \{T(Z) \wedge M(V)\};$

represents the expression

$[P(x) \vee Q(V)] \wedge R(x,y) \wedge [S(Z,x) \vee [T(Z) \wedge M(V)]]$

in fully parenthesized notation.

The only constructs whose meaning requires special explanation are <functional term>, <literal element>, and the connectives "&" and "⊕".

If a term is in the scope of the modifier "EV" then all functions in that term are applied to their arguments (i.e. evaluated as LISP functions) when that literal is used in the problem-solving process. "EVN" further specifies that the functions to be evaluated have numerical values. The default convention is that the term is manipulated as an unevaluated symbolic expression. The "REQUEST" modifier, which takes a literal as its argument, alters the way that literal is treated by the problem solver. This is discussed in Section 4.

The AND connective is denoted by "∧" . Thus a state satisfies the assertion A∧B if it satisfies both A and B. The weaker THAND connective is denoted by &. Exclusive OR is denoted by "⊕".

3.1.2 STATE DESCRIPTIONS: Assertions specifying states are restricted to be conjunctions of literals.

3.1.3 AXIOMS: Axioms are stated in either of the forms P⊃Q or P, where P and Q are assertions. They hold in all states and are used to complete a given state description by deduction of other elements of a state from those given.

3.1.4 RULES: There are three types of rules: primitive procedures, definitions, and iterative rules.

(a) A primitive procedure is specified by a name, an argument list, and its pre and post-conditions, i.e.

P {f (x_1 ,...,x_k)}Q where P and Q are assertions in which x_1 , ..,x_k are free, and f is the procedure name.

The variables are formal parameters of the procedure. They may be "bound" by substitution of actual parameters when the procedure is applied to a state.

For example consider the operator,

move(R1,O1,L1,L2):"R1 makes O1 from L1 to L2";
with preconditions,
ROBOT(R1) ∧ MOVABLE(O1) ∧ AT(O1,L1) ∧ AT(R1,L1) ∧¬ ON(R1,O2,L1);
and postconditions,
AT(O1,L2) ∧ AT(R1,L2);

When a primitive procedure is defined it may be declared to be an ASSUMPTION. If it is used in a successful program construction, then the user is informed and is given the opportunity to carry out a structured program development of this non-primitive operation. This is described in Section 7.

(b) A definitional rule is of the form R≡S where R and S are assertions. The relation, S, is given as the postcondition of the rule. The meaning of a definition is that whenever it is desired that S be true it is equivalent to establish the truth of R. A definition is often used to shorten assertions in rules by defining a single relation as equivalent to an often used condition.

(c) Iterative rules specify conditions that if satisfied justify the assembly of a "while" loop to achieve the associated goal. They are instances of the iterative rule S2 in Section 2.2, and are defined by giving:

(i) A name, e.g. TLOOP, (without parameters).
(ii) A basis assertion P.
(iii) A loop invariant assertion Q that specifies relations that must be true in the state prior to each iteration.
(iv) An iteration step assertion R that specifies the goals to be achieved during an execution of the loop body.
(v) An iterative goal G, the assertion considered achievable by the iterative process.
(vi) The format of iterative rules also allows the specification of a loop control test L and an output assertion S if they differ from G.

The rule,

TLOOP
P;Q;R;G;L;S;
where P,Q,R,G,L and S are assertions,
defines the iterative rule "TLOOP"
associated with the goal G.

3.1.5 SPECIAL AXIOMS: After the rules and initial state have been defined the system requests the following information for each predicate symbol P that has been mentioned. The system use of this information is discussed in Section 4.

a) "Is P a function of the state?" The intent of this classification is to separate those relations whose truth value may be affected by a state transformation, i.e., FLUENT relations,from those whose truth value is constant over all achievable worlds, i.e., NON-FLUENT relations such as "ROBOT(X)", "INTEGER(Y)".

b) "Is knowledge represented using P partial?" A partial relation may have truth values TRUE, FALSE, or UNDETERMINED. Partial relations may be used to represent incomplete knowledge of the world which may cause conditional statements to be generated as explained in Section 5. A relation may be declared "uncertain" which implies an absence of knowledge about it so that it is assigned a truth value of undetermined a priori. If P is not "partial" it is "total" and can only have truth values of either true or false. Thus rule R6 applies to partial predicates only.

c) "Does P have a uniqueness property in certain argument positions?" A "yes" answer indicates that P cannot be true for two sequences of argument values that differ only at one of those positions that are unique. The unique positions are given using the notation, (X1,*,X3,*,...,Xn), for

example, to designate the second and fourth argument positions. For each unique argument position in relation P(a1,...,an), an axiom is "built-in" from which a contradiction may be established with P(b1,...,bn) that differs in a unique position and matches elsewhere.

The statement, "an object can only be in one place at one time", is expressed by, AT(X1,*). If we add, "and only one object can be at any place", then we use AT(*,*).

3.1.6 SIMPLIFICATION: Algebraic simplification rules may be given to simplify the terms that may occur in subgoals during the problem solving phase. The simplification is driven by a table of rules of the form s=t where s and t are terms; occurrences of $s\alpha$ are replaced by $t\alpha$ for any substitution α.

The output format of any functional term may be specified by the user by giving a rule in which its input prefix form is on the left, e.g., (PLUS X Y) = (X+Y).

COMMAND SYNTAX	ACTION PERFORMED
TRY <rule1> BEFORE <rule2>	Use <rule1> before <rule2> to develop a subgoal.
FOR <rule> [FIRST] TRY <literal>	Change the precondition Q of <rule> to <literal> & Q if "FIRST" is given otherwise Q ∨ <literal>.
DELETE {<rule>,<literal>, <advice num>}	If <rule> is given, remove that rule. If <literal> then alter state to make <literal> not true. If <advice num> then delete the associated advice and undo its effects on the system.
ADD{<rule>,<literal>}	If <rule> is given then accept a new rule. If <literal> then alter state to make <literal> true.
ALTER <rule>	<rule> may be modified.
ASSUME {<rule>,<literal>}	If <rule> is given then an assumed rule may be defined. If <literal> then alter state to make <literal> true and mark it as an assumption.
RESTRICT <rule>{TO,FROM} <rule list>	For any goal in Q, if "TO" is given then only rules in <rule list> may be used, if "FROM" then no rule in <rule list> will be used.
ADVICE	All advice given that session is displayed.
STATUS	The following information is displayed: -rules entered and goals pending in current subgoal tree, -rules and goals in longest path obtained so far, -currently constructed program segment -longest program segment constructed so far.
PAIRWISE INEQUALITIES <proc>	Pairwise equality is prohibited in primitive procedure argument positions containing "*".
RECURSIVE <rule>	The rule may be used directly to achieve a goal in its pre-condition, otherwise it may not.

Figure 4

3.2. ADVICE LANGUAGE

The advice facility is intended to enable the user to impose structure relevant to solving a particular problem upon an already defined frame. This additional structure includes preference orderings among goals and rules, and restrictons on the search space. The preferences may also reflect the kind of solution the user wants.

Advice is given during program generation by means of an interactive facility. The advice subsystem may be entered by responding to a system query, "DO YOU HAVE ADVICE?" , or by typing any key during program generation. The user may request to see the current path in the subgoal tree i.e. rules entered and goals pending, and receive a diagnosis of the cause of any failure. This is useful in deciding what advice to give.

The advice system enters a read loop recognizing and numbering commands from the language shown in figure 4. In the language syntax, optional symbols are enclosed in "[" and "]"; enclosing a list of symbols in "{" and "}" indicates that one must be chosen; <rule> is a rule name; <rule list> is a list of rule names; <proc> is a primitive procedure name; <advice num> is of the form "#n", where n is an integer; and Q denotes the pre-condition of <rule>.

After advice has been given the system may be directed to reject the rule it is currently using, if any, or to try (perhaps re-try) the current rule.

The advice facility is an important tool for experimenting interactively with different frames to determine their adequacy and soundness. At present, the language is rudimentary and should be extended.

3.3 PROGRAMMING LANGUAGE

The generated programs are expressed in an elementary ALGOL-like language

which includes block structure, assignment statements, conditional statements, while loops, and non-recursive procedures calls. The procedures may be typed, including Boolean, and may have side effects in addition to the value returned. The procedure parameters are normally called by value except in the case of special W-variables in conditional assignments (rule R0,section 2).

3.4 AN EXAMPLE

Consider the task of writing a program to compute the nth Fibonacci number for some integer n. This task has been posed in [Balzer 1972]. The basic information required is the recursive definition and the basis values. One way to express this in the Frame language uses the following predicates with the indicated meanings:

VFIB(X,Y): "The value of the X Fibonacci number is Y",
C(X,Y): "The contents of the variable X is Y",
FIB(X,Y): "The variable X contains the Y Fibonacci number,
INTEGER(X): "X is an integer",
ISVAR(X): "X is a variable",
>(X,Y): "X is greater than Y"
NEWVAR(X,Y): "X and Y are local variables".

The problem is ISVAR(X3)∧INTEGER(N){?}FIB(X3,N).

The frame contains:

1. Axioms VFIB(1,1)and VFIB((ADD1 1),2)(these define initial values).

2. Axiom
   TAFIB
   VFIB((SUB1 V1),V2)∧VFIB((SUB1(SUB1 V1)),V3)∧ =(V4,(PLUS V2 V3));
   VFIB(V1,V4);
   (defines VFIB(V1,V4) for terms beyond the initial values).

3. An iterative rule (named TFIB) with goal FIB(X3,N); this rule defines the conditions to be satisfied during an iterative upward computation. The basis condition (to initialize the counter and program variables) is:

   NEWVAR(V1,V2)∧INTEGER(V8)∧C(V1,(ADD1 1))∧C(V2,1)∧C(V3,(ADD1 1));.

The loop invariant condition is:

   C(V1,V5)∧C(V2,V9)∧C(V3,V10)∧VFIB(V5,V10)∧VFIB((SUB1 V5),V9);.

This states that at each entry to the loop body, if the value in the counter is i and the values in the program variables are j and k then j is the ith Fibonacci number and k is the (i-1)st Fibonacci number.

The iteration step condition

C(V1,(ADD1 V5))∧FIB(V2,V5)∧FIB(V3,(ADD1 V5));

specifies what the iteration step is to accomplish. The control test, >(V5,V8) and an output assertion FIB(V3,V8) are given.

4. A definition of FIB in terms of VFIB and C

TDFIB
VFIB(V2,V3)∧C(V4,V3); FIB(V4,V2);

5. A simple primitive procedure for assignment is also given, i.e.

←(V1,A1)
ISVAR(V1); C(V1,A1);.

No rules are declared as assumptions. The additional information given to complete the frame specification is shown in figure 5, and a program generated from this frame is shown in figure 6.

PREDICATE SYMBOL	FLUENT	PARTIAL	UNIQUENESS
C	TRUE	FALSE	C(X,*)
FIB	TRUE	FALSE	FIB(X,*)
>	TRUE	FALSE	FALSE
VFIB	TRUE	FALSE	VFIB(*,*)
INTEGER	FALSE	FALSE	FALSE
=	TRUE	FALSE	FALSE
ISVAR	FALSE	FALSE	FALSE

SIMPLIFICATION RULES:

(ADD1 (SUB1 X)) → X

(SUB1 (ADD1 X)) → X

FUNCTION OUTPUT SYNTAX:

(ADD1 X) = (X+1)

(SUB1 X) = (X-1)

(PLUS X Y) = (X+Y)

ADVICE: TRY TFIB BEFORE TDFIB

RECURSIVE TAFIB

Figure 5

```
PROC1 (X3,N)
ISVAR(X3);INTEGER(N);
COMMENT
INPUT ASSERTION
NONE
OUTPUT ASSERTION
FIB(X3,N)
   BEGIN
   Y1 ← (1+1);
   Y2 ← 1;
   X3 ← (1+1);
   WHILE ¬>(Y1,N) DO
      bEGIN
      Y1 ← (Y1 + 1);
      Z2 ← X3;
      X3 ← (X3 + Y2);
      Y2 ← Z2;
      END
   END
```

Figure 6

4. PROBLEM SOLVING PROCESSES

During the process of problem solving and program generation, information is needed at many points to reduce the search space or to produce reasonable programs. Some of the information is provided in the frame specification by statements about the rules and predicates; other useful facts are provided to the problem solver in the form of rather simple advice. Roughly speaking, there are six basic processes in the problem-solving system where extra facts can help: (a) pattern matching, (b) development of nodes in the subgoal tree, (c) updating the state description (i.e. implementing invariance), (d) backtracking in the subgoal tree, (e) conditional branching, (f) assembly of programs. Each fact (as opposed to a rule or axiom) in a frame specification and each sort of advice has at least one function in speeding up a basic process. Below we describe some of the ways in which the present variety of facts and advice is used.

(1) OR-Node Selection. When more than one rule can be applied to reduce a given goal, some selection and preference criteria are needed. By using the advice system,the rules and axioms that may be applied to achieve goals within the precondition of a rule or axiom may be restricted to or excluded from a given list. Also, a preference ordering may be specified among rules and axioms with common post-conditions. Goals within the preconditions of axioms are always restricted to deduction within the current state, i.e. can be reduced only by use of other axioms, and do not cause a state transformation nor add any construct to the generated program.

(2) Predicate Classification. A predicate P is classified according to the kind of subgoaling permitted to achieve a goal of the form P(t). If P is declared to be NON-

FLUENT, then any goal literal containing P can be achieved only by deduction from the current state. No rules (procedure, iterative or definitional) are applied. FLUENT goals are attempted by deduction and state transformation. If a fluent predicate occurs in a literal which is the argument of the REQUEST modifier, then it is treated as a non-fluent.

(3) Goal Ordering. The achievement of a condition (and the efficiency of the output program) is strongly influenced by the ordering of its subgoals. In particular, the bindings of variables occurring in goals may be determined by earlier achieved instances. In some cases only certain orderings will permit achievement. An objective of an automatic problem solving system is to determine the optimal subgoal ordering, but at present this is provided by the user when the Frame is defined and may be altered by advice. However, the system automatically orders non-fluent goals first in a condition; this relatively short achievement search is used both as a quick rejection strategy and to get variable bindings of the correct type for the remaining fluent goals.

(4) Recurring failures. When failure occurs in some subtree prior to successfully solving a subproblem, its causes should be used to avoid repeating the same failure in the continued search if possible. At present this must be handled using the interactive advice system. This informs the user of the current path in the subgoal tree, current program generated, and goals that fail, thus allowing interactive correction when a repetition occurs. These situations can also be eliminated by placing the (eventual) successful subprograms on the program library for use as MACROS.

(5) Repetition. Certain types of looping behavior in the subgoaler are prevented using the feature of the Frame language that allows a rule to be declared recursive or non-

recursive. If declared non-recursive, then that rule will not be used directly to achieve a goal in its pre- condition and it will not be entered twice to achieve the same instance of its post-condition within the same subgoal tree. A more general mechanism should consider not only the current goal and rule but also the current state as well.

(6) Truth Values. Though the underlying semantics is three valued, search efficiency is gained by restricting relations involving certain predicate symbols to be two valued. If a predicate P is declared to be TOTAL, then failure to achieve P indicates that ¬P is true. Only true positive instances of total predicates are stored in the state. The rule of undetermined values is not applicable to literals involving total predicates. The additional processing required for PARTIAL predicates is described in Section 5.

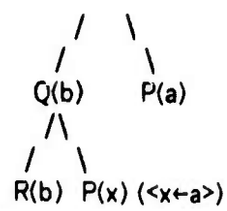
(7) Useless Procedure Calls. In some cases, the application and generation of redundant or trivial procedure calls are detected and avoided. At the moment this is done by placing restrictions in the frame on the actual parameters of primitive procedures. The system will not use an instance of a primitive procedure that contains pairwise equality between its actual parameters that has been prohibited by the user. For example, the advice "PAIRWISE EQUALITY MOVE(x1,x2,*,*)" will cause the rejection of the procedure call "MOVE(MAN,CHAIR,P,P)".

(8) Uniqueness Properties. Uniqueness or single-valuedness in argument positions of certain predicates is sufficiently important to justify a special mechanism rather than to rely on deduction using axioms. The designation of certain argument positions as unique is equivalent to efficiently building in axioms of a particular form, e.g. P(x1,*) represents the axiom,

$$P(x1,x2) \wedge x2 \neq x3 \rightarrow \neg P(x1,x3).$$

These special axioms are used for consistency checking (in the implementation of the rule of invariance) when the state is updated.

(9) Context Linking. The context, which includes the state and bindings on subgoals currently pending at a node, should be available to aid search decisions, e.g. instantiations of subgoals or choice of rule, at descendent nodes in the subgoal tree. The system has a mechanism that if requested will keep track of the instantiated goals at each level of the subgoal tree so that their variable bindings are available when attempting lower level goals that precede them in the depth first ordering. This is used to instantiate the lower level goals. For example, suppose Q(b) ∧ P(a) is a condition to be achieved and a primitive procedure R(y) ∧ P(x) {p(x,y)}Q(y) is applied to achieve Q(b), then for the P(x) in the precondition of p, P(a) will be used since it must be achieved at the higher level anyway, i.e.,



This heuristic may be viewed as the opposite of subsumption, the strategy being to get ground instances as soon as possible to help avoid long searches using rules. This is a rather restrictive strategy that may exclude solutions and is only used when requested by the user.

(10) Evaluation of Predicates and Functions. For certain predicates occurring in subgoals, achievement is most efficient by direct evaluation. If a literal occurring in a goal is formed with a predicate that has a LISP definition, then that literal is evaluated as a LISP statement. Special processes or even subsystems can thereby be linked into program generation. Evaluation of arbitrary functions occurring in terms in arguments of goal literals is done if the function occurs in the scope of an EV modifier. These evaluations assume the soundness of implicit axioms describing the LISP definitions,

and the consistency of these axioms with the Frame. For example, the equality predicate, "=", is evaluated using the LISP "EQUAL", and the predicate NEWVAR(x1,x2,...,xn) takes an arbitrary number of arguments and binds each Frame variable xi to a new program variable (for use perhaps as a local variable in a block).

(11) Simplification rules. Rules of the form s → t where s and t are terms, may be included in the Frame. Such rules are applied to simplify terms in goals by replacing occurrences of sα by tα. This not only reduces the complexity of terms in the subgoal tree, but it also modifies the pattern matching process and the set of rules that can be applied to reduce a goal.

(12) Computing Input/Output Assertions. In Section 2 primitive procedures were viewed as Frame rules of the form ||-P{p}Q, where P and Q are the pre and postconditions for p. The conditions P and Q may also be viewed as sufficient input and output assertions for p , that must be satisfied by the actual parameters of p. For any generated program segment A, the input assertion I_a is computed as the conjunction of all literals, l, from a state that were used in achieving subgoals encountered during the generation of A and did not occur in that state as a result of a postcondition of a procedure whose generation in A preceded the addition of l to I_a. The output assertion O_a is the conjunction of literals added to a state during the generation of A that are true in the final state. The usefulness of computing sufficient input and output assertions for a program or segment thereof will become apparent when we discuss program generalization and the construction of conditional statements.

All of these applications of facts and advice with the exception of (12), are intended to have a direct effect on reducing the growth of the subgoal tree (process

(b)). In addition, the pattern matching process (a) is extended by (11); (c) is aided by the restriction of truth values and the special axioms (6,8); (e) is dependent on (6 and 12); (f) is aided by (3,7,11,12). There are other techniques, mainly details of the implementation, some of them heuristic, that affect problem solver, particularly the backtrack (d), the updating (c) and assembly of programs (f) (e.g. the implementation of the $\wedge$ connective by software interrupts that protect already achieved goals, includes certain assumptions about backtracking when an AND-node fails).

5. GENERATION OF CONDITIONAL STATEMENTS

Conditional statements are generated in situations where the rule of undetermined values (R6) applies or when the outcome of a primitive procedure is uncertain. In this section the system methods for constructing conditionals will be described and an example given. The question of extending the formal algorithm and the correctness proof is considered.

5.1 UNCERTAIN PRECONDITIONS

As previously mentioned, relations involving partial predicates may have truth values of TRUE, FALSE, or UNDETERMINED, whereas all other relations must be either TRUE or FALSE. Partially valued predicates are intended to express the possibility of an uncertainty or lack of knowledge about a state arising during the problem solving and program generation phase of the system. The formal algorithm for deciding when an uncertainty has arisen is rule R6. As with invariance, the implementation of R6 is only an approximation to the formal rule. The system may give up too early, but this, in itself, does not lead to incorrect programs, merely redundant ones.

5.1.1 UNDETERMINED VALUES. During the generation of a program, uncertainty may arise when a precondition for the application of a rule is UNDETERMINED with respect to the current state. The implementation of the rule R6 is described by the following definitions:

DEFINITION A literal l is UNDETERMINED in a state S if the following conditions hold:

(i) pred(l) is partial,
and (ii) the system halts without solving $S\{?\}l$,
and (iii) the system cannot prove $S \cup F \supset \neg l$.

Condition (ii) means that l is not true in S nor can S be transformed into a state

in which l is true. If condition (ii) is true and ¬l is true in S then l must retain a truth value of FALSE and the precondition subgoal l must fail. Failure to prove ¬l from S establishes a truth value of UNDETERMINED for l with respect to S. This definition applies to fluent and nonfluent literals but since the truth value of a "nonfluent" cannot be changed by a state transformation, for them, it is sufficient to use only the logical axioms in deciding condition (ii).

For the more general case in which the precondition may be a disjunction of literals we have the definition,

DEFINITION A disjunction of literals $\{l_i\}_{i=1}^{n}$ is UNDETERMINED in a state S if at least one literal is UNDETERMINED and no literal can be achieved from S.

5.2 CONDITIONAL STATEMENTS

When a pre-condition P is UNDETERMINED in a state S, a conditional branch is inserted in the solution program. If P is a single literal l, then program generation may continue either along the path in which l is assumed to be TRUE and in which future goals are attempted with respect to state S U{l}, or along the path in which ¬l is assumed to be TRUE using state S U{¬l}. The system convention has been to generate a call to a yet ungenerated procedure for the latter case. The tasks of generating such contingency programs are placed in a subproblem stack for later attention (see Section 5.5). Program generation continues, by convention, along the path using state S U {l}. This path is referred to as the "trunk" program of the tree of contingency programs generated while attempting to achieve the main goal. The path selection at present is rather ad hoc since no assignments of probability are made at the points of uncertainty. For an undetermined disjunction $\{l_i\}_{i=1}^{n}$.

```
if ¬l1     then
  if ¬l2     then
   .
    .
     .
      if ¬lm     then pm
      else pm-1
     .
    .
   .
   else p1
else p0
```

where each p_i is a call to a program to achieve a selected goal G

from state $S_i = S \wedge \{l_i : i=j+1 \ \& \ i<n\} \wedge \{\neg l_i : 1 \le i \le j\}$ and p_0 is the trunk program segment which satisfies $S \wedge l_1 \{p_0\} G$ and forms the else-statement in the main-

clause of the conditional. Each member of the set of triples $\{(p_j , S_j ,G):1\leq j\leq m\}$ is placed in the stack of contingencies and program generation continues for p_θ The assumed literal, l_1, is removed from the state following the generation of the ELSE clause in the trunk program if it is not in the output assertion.

5.3 SELECTION OF CONTINGENCY GOAL

The goal G to be achieved by the contingency programs is selected from the set of goals in the subgoal tree that are global to the undetermined precondition. Let us refer to the set of goals which are below G in the subgoal tree, as the SCOPE of G. The particular G chosen and its associated scope affect the length of p_θ , duplication among contingency programs, degree of difficulty in generating contingency programs and validity of their use. If the structure of the trunk program is to remain fixed during contingency program generation then the choice of G cannot be deferred. The block structure of our program language imposes the restriction that for any conditionals in p_θ, a contingency goal G' must not have a greater scope than G. There is also the problem that if G is not fully instantiated then inconsistent instantiations may occur in different contingency programs which must validly rejoin the main program following the ELSE clause. The present system selects the least global fully instantiated goal thereby satisfying the block nesting constraint and minimizing the scope while avoiding the problem of handling deferred instantiation. This selection process is always effective in the present system since the top level goal is fully instantiated.

5.4 REJOIN CONDITIONS

When a contingency program is generated its output state must satisfy certain conditions, hereafter called the rejoin condition, for return of control to the trunk program to be correct. Consider the case of an undetermined goal L in state S and a contingency goal G in figure 7 . Let A and B be program segments that satisfy $S \wedge L\{A\}G$ and $S \wedge \neg L\{B\}G$ and let C be the rest of the trunk program.

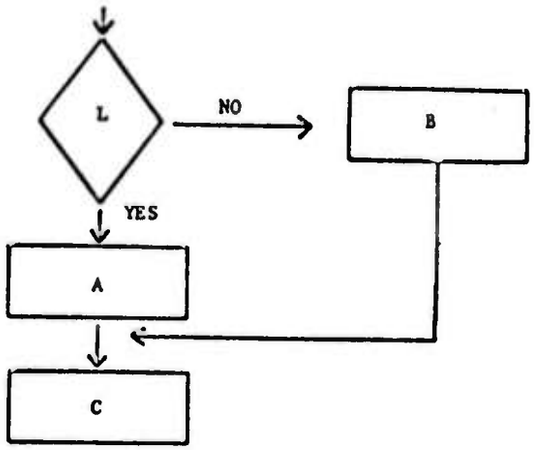


Figure 7

Let R be the output state of B obtained by applying invariance; thus $S \wedge \neg L\{B\}R$ and $R \supset G$. Similarly, let $S \wedge L\{A\}P$ where $P \supset G$, and let Q be the sufficient subset of P required as input to C (see Section 4(12)). Then, the REJOIN CONDITION for B is $R \supset Q$. B is said to have BAD SIDE EFFECTS if in fact $R \supset Q$ cannot be established.

5.5 SUBPROBLEM STACK

The task of generating a contingency procedure is specified by the quadruple:

(<procname> <state> <goal> <rejoincond>)
where,
<procname> is the name of the yet ungenerated procedure that must satisfy <state>{<procname>}<goal> ∧ <rejoincond>.

At the point in the planning when the uncertainty is encountered, the first three elements of the quadruple are placed in a stack. The rejoin condition is not known at this time since it involves the input assertion for the trunk segment C following the point where control returns from the contingency plan to the trunk plan. After C is generated, the rejoin condition is computed and stored as the fourth element of the quadruple.

When planning has been completed for a trunk procedure, if the subproblem stack is not empty then contingency planning may be done by removing a quadruple from the stack and posing this as a program generation task. The state of the system is initialized to the specified contingency state and the subgoaling system is given <goal> as its main goal. If it is successful in achieving a state in which the main goal is true then a test is made to see if the rejoin condition is true in that state. If it is then the procedure declaration is adjoined to its trunk program. If the condition cannot be proved, the system allows the user two alternatives: (i) Mark the call to the program as an error exit in the trunk program, or (ii) "Fit" the program to the trunk program by posing the currently untrue rejoin condition as a new goal, constructing a new program segment that achieves it, and appending this segment to the end of the contingency program.

This process of generating a trunk procedure which may create new contingency

tasks then generating contingency procedures as directed by the user may continue until all contingencies have been processed and the stack is exhausted.

5.6 COMPUTATION OF INPUT-OUTPUT ASSERTIONS

The computation of input-output assertions for programs not containing conditionals is described in Section 4(12). The uncertainty as to which path computation will follow in a program containing conditional statements complicates these assertions. The input-output assertions in this case must be computed incrementally as each contingency program is generated.

In the conditional statement shown in figure 7, suppose we know the minimal input and output assertions for A and B, say P{A}Q and R{B}S. then the input and output assertions for the conditional statement are

$$(L \wedge P) \vee (\neg L \wedge R)\{\text{if } L \text{ then } A \text{ else } B\}Q \vee S.$$

To reduce computation, We use the simpler sufficient input assertion $P \wedge R$, (Note that $P \wedge R$ should be consistent since it is a subconjunct of a previous state). There doesn't appear to be a simplifying approximation for output assertions .

5.7 UNCERTAIN PRIMITIVE PROCEDURES

A primitive procedure q defined by P{q}Q has an uncertain outcome if Q is a disjunction. In the present system, disjunctive post-conditions use the exclusive OR connective, "⊕". This allows us to define frame procedures that have an intended result but may be unreliable. It is assumed that exactly one of the possible outcomes will be true in the output state . At the point where an uncertain operator is applied, the problem solver has no knowledge of what the outcome will be and a conditional statement must be generated. Let Q be the disjunction of literals $\{l_i\}_{i=1}^{n}$. The first outcome l_1 is considered to be the normal (goal) result of executing q. Following the inclusion of q in the program in state S, a conditional statement of the following form is generated.

if $\neg l_1$ then

if $\neg l_1 \wedge l_2 \wedge \neg l_3 \wedge \ldots \wedge \neg l_n$ then p_2

else if $\neg l_1 \wedge \neg l_2 \wedge l_3 \wedge \neg l_4 \wedge \ldots \wedge \neg l_n$ then p_3

.

.

.

else if $\neg l_1 \wedge \neg l_2 \wedge \ldots \wedge \neg l_{n-1} \wedge l_n$ then p_n

else p_{n+1}

where each p_j, $2 \leq j \leq n$, is a call to a program to achieve l_1 from state $S_1 = S \cup \{l_1\} \cup \{\neg l_i : i \neq j \ \& \ 1 \leq i \leq n\}$ and p_{n+1} is an error exit. The contingency states will correspond to the n ways of assigning exactly one literal true and the remaining literals false.

5.8 AN EXAMPLE

Suppose a procedure is to be generated for a man to travel from San Francisco to New York given three modes of travel, i.e., flying, driving, or walking. This is similar to the "airport problem" discussed in [McCarthy 1959]. A FRAME for this problem consists of defining a primitive procedure for each mode of travel, an initial state, and relation information as shown in figure 8. A few of the contingency programs generated are shown in figure 9.

RELATIONS	DEFINITION	FLUENT	PARTIAL	UNIQUENESS
ROB(X)	"X is a robot"	FALSE	FALSE	FALSE
AUTO(X)	"X is an automobile"	FALSE	FALSE	FALSE
PLANE(X)	"X is an airplane"	FALSE	FALSE	FALSE
AIRPORT(X)	"X is an airport"	FALSE	FALSE	FALSE
AT(X,Y)	"X is at location Y"	TRUE	FALSE	AT(X,*)
WALKABLE(X,Y)	"A walkable path exists between X and Y"	TRUE	TRUE	FALSE
CLEAR(X,Y)	"The sky is clear between X and Y"	TRUE	TRUE	FALSE
DRIVABLE(X,Y)	"A drivable road exists between X and Y"	TRUE	TRUE	FALSE
HASUMBRELLA(X)	"X has an umbrella"	TRUE	TRUE	FALSE
CRASHED(X,Y,Z)	"X crashed between Y and Z"	TRUE	FALSE	FALSE
KILLED(X)	"X has been killed"	TRUE	FALSE	FALSE
RUNS(X)	"X will run properly"	TRUE	TRUE	FALSE
FLIES(X)	"X will fly properly"	TRUE	TRUE	FALSE

****

PRIMITIVE PROCEDURE	PRE-CONDITIONS	POST-CONDITIONS
walk(R1,L1,L2) "R1 walks from L1 to L2"	ROB(R1)∧¬ KILLED(R1)∧AT(R1,L1) ∧CLEAR(L1,L2)∨HASUMBRELLA(R1) ∧WALKABLE(L1,L2);	AT(R1,L2)
drive(R1,C1,L1,L2) "R1 drives C1 from L1 to L2"	ROB(R1)∧¬ KILLED(R1)∧AUTO(C1) AT(C1,L1)∧RUNS(C1) ∧DRIVABLE(L1,L2)∧AT(R1,L1);	AT(R1,L2) ∧AT(C1,L2)
fly(R1,A1,L1,L2) "R1 flies A1 from L1 to L2"	ROB(R1)∧¬ KILLED(R1)∧PLANE(A1) ∧AIRPORT(L2)∧AT(A1,L1) ∧FLIES(A1)∧CLEAR(L1,L2) ∧AT(R1,L1);	[AT(R1,L2)∧ AT(A1,L2)] ⊗[CRASHED(A1,L1,L2) ∧KILLED(R1)]

****

INITIAL STATE

ROB(MAN)∧AUTO(BMW)∧PLANE(F111)∧AIRPORT(SFO)∧AIRPORT(NYC)∧AT(MAN,HOME)∧AT(BMW,GARAGE)∧AT(F111,SFO);

****

ADVICE

PAIRWISE INEQUALITIES: walk(R1,*,*),drive(R1,C1,*,*),fly(R1,A1,*,*)

TRY FLY BEFORE DRIVE, TRY DRIVE BEFORE WALK

Figure 8

```
PROC1(MAN NYC)
ROB(MAN);AUTO(BMW);PLANE(F111);AIRPORT(NYC,);
COMMENT
INPUT ASSERTION:
AT(MAN HOME)^CLEAR(HOME GARAGE)^AT(BMW GARAGE)^AT(F111 SFO)
^FLIES(F111)^CLEAR(SFO NYC)^RUNS(BMW)
^DRIVABLE(GARAGE SFO)^WALKABLE(HOME GARAGE)
OUTPUT ASSERTION:
AT(BMW SFO)^AT(F111 NYC)^AT(MAN NYC);
COMMENT
PROC11(ATTEMPTS_TO_ACHIEVE_(AT(MAN NYC)
PROC10(ATTEMPTS_TO_ACHIEVE_(AT(MAN GARAGE)
PROC9(ATTEMPTS_TO_ACHIEVE_(AT(MAN GARAGE)
PROC8(ATTEMPTS_TO_ACHIEVE_(AT(MAN GARAGE),
PROC5(ATTEMPTS_TO_ACHIEVE_(AT(MAN SFO)
PROC4(ATTEMPTS_TO_ACHIEVE_(AT(MAN SFO)
PROC3(ATTEMPTS_TO_ACHIEVE_(AT(MAN NYC)
PROC1(ATTEMPTS_TO_ACHIEVE_(AT(MAN NYC),;
   BEGIN
   IF ¬FLIES(F111) THEN
      PROC1(MAN NYC)
   ELSE
      BEGIN
      IF ¬CLEAR(SFO NYC)) THEN
         PROC3(MAN NYC)
      ELSE
         BEGIN
         IF ¬RUNS(BMW)) THEN
            PROC4(MAN SFO)
         ELSE
            BEGIN
            IF ¬DRIVABLE(GARAGE SFO)) THEN
               PROC5(MAN SFO)
            ELSE
               BEGIN
               IF ¬CLEAR(HOME GARAGE)) THEN
                  IF ¬HASUMBRELLA(MAN)) THEN
                     PROC8(MAN GARAGE)
                  ELSE PROC9(MAN GARAGE)
               ELSE
                  BEGIN
                  IF¬WALKABLE(HOME GARAGE)) THEN
                     PROC10(MAN GARAGE)
                  ELSE
                     BEGIN
                     WALK(MAN HOME GARAGE)
                     END
                  END
               DRIVE(MAN BMW GARAGE SFO)
               END
            END
         FLY(MAN F111 SFO NYC)
         IF ¬AT(MAN NYC) THEN
            IF ¬AT(MAN NYC) ∧ CRASHED(F111 SFO NYC)
               PROC11(MAN NYC)
            ELSE PROC1ER(MAN NYC)
      END END
   END

PROC2(MAN NYC)
ROB(MAN);AUTO(BMW);
COMMENT
INPUT_ASSERTION:
AT(MAN HOME)^CLEAR(HOME GARAGE)^AT(BMW GARAGE)^RUNS(BMW)
^DRIVABLE(GARAGE NYC)^WALKABLE(HOME GARAGE)
```

Figure 9

```
OUTPUT_ASSERTION:
AT(BMW NYC)∧AT(MAN NYC);
COMMENT
PROC16 ATTEMPTS_TO_ACHIEVE_ (AT MAN GARAGE)
PROC15 ATTEMPTS_TO_ACHIEVE_ (AT MAN GARAGE)
PROC14 ATTEMPTS_TO_ACHIEVE_ (AT MAN GARAGE)
PROC13 ATTEMPTS_TO_ACHIEVE_ (AT MAN NYC)
PROC12 ATTEMPTS_TO_ACHIEVE_ (AT MAN NYC);
   BEGIN
   IF ¬RUNS(BMW) THEN
      PROC12(MAN NYC)
   ELSE
      BEGIN
      IF ¬DRIVABLE(GARAGE NYC) THEN
         PROC13(MAN NYC)
      ELSE
         BEGIN
         IF ¬CLEAR(HOME GARAGE) THEN
            IF ¬HASUMBRELLA(MAN) THEN
               PROC14(MAN GARAGE)
            ELSE PROC15(MAN GARAGE)
         ELSE
            BEGIN
            IF ¬WALKABLE(HOME GARAGE) THEN
               PROC16(MAN GARAGE)
            ELSE
               BEGIN
               WALK(MAN HOME GARAGE);
               END
            END
         DRIVE(MAN BMW GARAGE NYC)
         END
      END
   END


PROC4(MAN SFO)
ROB(MAN);
COMMENT
INPUT_ASSERTION:
AT(MAN HOME)∧CLEAR(HOME SFO)∧WALKABLE(HOME SFO)
OUTPUT_ASSERTION:
AT(MAN SFO);
COMMENT
PROC25 ATTEMPTS_TO_ACHIEVE_ (AT MAN SFO)
PROC24 ATTEMPTS_TO_ACHIEVE_ (AT MAN SFO)
PROC23 ATTEMPTS_TO_ACHIEVE_ (AT MAN SFO);
   BEGIN
   IF ¬CLEAR(HOME SFO) THEN
      IF ¬HASUMBRELLA(MAN) THEN
         PROC23(MAN SFO)
      ELSE PROC24(MAN SFO)
   ELSE
      BEGIN
      IF ¬WALKABLE(HOME SFO) THEN
         PROC25(MAN SFO)
      ELSE
         BEGIN
         WALK(MAN HOME SFO);
         END
      END
   END

PROC12(MAN NYC)
ROB(MAN);
COMMENT
INPUT_ASSERTION:
AT(MAN HOME)∧CLEAR(HOME NYC)∧WALKABLE(HOME NYC)
```

Figure 9 - continued

```
OUTPUT_ASSERTION:
AT(MAN NYC);
COMMENT
PROC30 ATTEMPTS_TO_ACHIEVE_ (AT MAN NYC)
PROC27 ATTEMPTS_TO_ACHIEVE_ (AT MAN NYC)
PROC26 ATTEMPTS_TO_ACHIEVE_ (AT MAN NYC);
   BEGIN
   IF ¬CLEAR(HOME NYC) THEN
      IF ¬HASUMBRELLA(MAN) THEN
         PROC26(MAN NYC)
      ELSE PROC27(MAN NYC)
   ELSE
      BEGIN
      IF ¬WALKABLE(HOME NYC) THEN
         PROC30(MAN NYC)
      ELSE
         BEGIN
         WALK(MAN HOME NYC)
         END
      END
   END
```

Figure 9 - continued

5.9 CORRECTNESS

Conditional statements will be correctly generated if the system methods are an accurate implementation of the conditional rule, R5, presented in Section 2. Referring to figure 7 in Section 5.4, if we let S be the output state of C then by construction and by verifying the rejoin conditions we have,

(1) $I \wedge L\{A\}G \wedge Q$,
(2) $I \wedge \neg L\{B\}G \wedge R$,
(3) $Q\{C\}S$,
(4) $\vdash R \supset Q$, (rejoin condition verification)

and the correctness argument may then be completed as follows,

(5) $I \wedge \neg L\{B\}G \wedge Q$, (2,4,Consequence Rule)
(6) $I\{$if L then A else $B\}G \wedge Q$, (1,5,Conditional Rule)
(7) $I\{$if L then A else B;C$\}S$, (3,6,Composition Rule).

It should be noted that if conditional statements occur in B then R may only be an approximation of the true output state resulting from executing B as discussed in Section 5.6. Similarly Q may be only an approximation of the true input assertion for the remainder of the program. In these cases an incorrect program may result. However the above argument serves as a justification for the system methods.

6. GENERATION OF ITERATIVE STATEMENTS

An iterative rule allows the program generator to construct a WHILE loop provided it can construct a loop body to satisfy the premisses of the rule. Ultimately such rules should require the user merely to specify an invariant in order to have the system write a correct iterative program. At the moment, the user needs to furnish some additional relevant facts. The algorithms used in the system to implement iterative rules of the form 32 (Section 2) and to assemble while loops are described briefly and an example given. Details of of the system implementation are found in Section 9.

6.1 PREMISSES FOR CONSTRUCTING A LOOP

An iterative rule is defined by the assertions P(basis), Q(loop invariant), R(iteration step goal), G(rule goal), L(control test) and S(output assertion). All the free variables in R and L must be among the free variables in Q. In order to use the rule, to achieve I{?}G say, the formal algorithm requires that all of the following subgoals be achieved or be true:

(i) Construct A such that L(F)||- I{A}P
(ii) L(F)|- I{A}Q
(iii) Construct B such that L(F)||-Q∧L{B}R
(iv) L(F) |- Q∧L{B}(∃Z)Q(Z)∨(¬(∃Z)Q(Z)∧¬L(Y))
(v) Construct C such that L(F) ||- Q∧L{B;C}Q∨-L

Note that (ii) and (iv) are restricted to first order rules (consequence, invariance, and the frame axioms). The input state for (iii) is Q∧L. In addition, an iterative rule must satisfy the following minimal consistency requirements within the frame F.

(vi) ¬(S ∪ F ⊃ L) and S ∪ F ⊃ G.

The conclusion of the rule is: I{A;WHILE L DO BEGIN B;C END}G.

Iterative frame rules are instances of the iteration rule [Hoare 1969]:

$$\frac{Q \wedge L\{A\}Q,\ Q \wedge \neg L \supset G}{\qquad\qquad}$$

Q{WHILE L DO A}G .

It is possible to derive a weak form of the rule:

$$\frac{Q \wedge L\{A\}Q \vee \neg L,\ \neg L \supset G}{Q\{\text{WHILE L DO A}\}G}$$

The weak form allows the invariant to fail on exit from the loop. We have found the weak form convenient to use in many examples.

The present implementation sets up clauses (i) - (iv) as a THAND of subgoals to be achieved. More specifically, suppose an iterative rule is invoked to solve the problem I{?}G. Let V be the list of variables in Q. The system does the following:

(1) A program segment p(P) is generated such that I{p(P)}I' and I'∪F |- P (p(P) may be empty).

(2) An instance Qλ of the loop invariant must be true in the state I', i.e. $\lambda = \{\langle v_1 \leftarrow s_1 \rangle,\ldots,\langle v_n \leftarrow s_n \rangle\}$ is constructed such that I'∪F ⊃ Qλ. (3) A program segment p(R) is generated such that Q ∧ L{p(R)}I" and I"∪F ⊃ R.

(4) It is checked that I"∪F⊃Q/β∨¬L for some substitution β and a set of conditional assignment statements C is constructed such that I"{C}Q ∨ ¬L.

Thus, at the moment, clause (iv) ensures that C need contain only conditional assignments. In the future we would want to relax this restriction. It is assumed that the user's definition of the rule satisfies (vi). The user may omit S or L; in the latter case ¬ G is used as the control test.

6.2 ASSEMBLY OF WHILE LOOPS

After the premisses have been achieved, a loop is assembled as follows:

(1) Let Y and W be two distinct lists of variables in one-to-one correspondence with V. For each $\langle v_i \leftarrow s_i \rangle \in \lambda$ construct an initial assignment statement "$y_i \leftarrow s_i$ ". Let "Y ← S" denote "$y_1 \leftarrow s_1 ; y_2 \leftarrow s_2 ; \ldots ; y_n \leftarrow s_n ;$".

(2) The WHILE loop may then be assembled in the form:

```
p(P);
Y ← S;
WHILE L(Y) DO
  BEGIN
  p(R(Y));
  IF Q(W) THEN Y ← W;
  END
```

where Q(W) is an expression containing calls to Boolean procedures indicated (syntactically) by the presence of the special W-variables (Section 2, Rule R0).

There are many heuristics in the system to reduce the number of program variables, i.e. y's and w's generated, to select the relevant portion of Q to be used in conditional assignment statements, to generate simple assignment statements (whose right hand sides are functional terms composed from functions in the frame) instead of conditional assignments, and to eliminate unnecessary assignment statements in the assembled program. These may all be classified as optimizations, some of which are done as the "WHILE" loop is assembled and others during a later optimization phase.

6.3 UPDATING THE STATE

After the while statement has been generated, the system updates the state. If an explicit output assertion S is given then the rule of invariance is applied in the same manner as with the postcondition of a primitive procedure. In the absence of an output assertion, a special update procedure runs the loop interpretively on the state

until the goal G becomes true. The resultant state is used in further planning. This latter method is useful when the global effects of the loop computation are so extensive, or even unpredictable, that an explicit specification of S is difficult. It may result in excessive update computation, particularly when loops are nested.

6.4 AN EXAMPLE

As an example of "while" loop generation consider the task of generating a program to compute the value of n factorial for some positive integer n where multiplication is not a primitive operation but is done by repeated addition. The Frame for this problem is shown in figure 10. Also used is the primitive procedure for assignment used in the example in Section 3. To achieve the goal "FACT(X0,N)" the system applies the iterative rule TFACT. The premises are achieved according to Section 6.1 which results in an application of another iterative rule TPROD. The premises of TPROD are achieved, the "inner" loop assembled and optimized and state is updated with respect to the output assertion. The assembled while loop is appended to the iteration step program for TFACT. The "outer" loop is then assembled and optimized and the state further updated reflecting the total state transformation of an execution of the nested loop program.

The output program after optimization with statements labeled according to their source of generaton in the algorithm is shown in figure 11. Note that successive values of the program variables are obtained by simple assignment statements rather than by conditional assignment as described in the algorithm. This is the result of applying system heuristics which are able to use the arithmetic operations PLUS and ADD1 which are primitive functions in the frame, to replace the conditional assignments.

RELATIONS	DEFINITION	FLUENT	PARTIAL	UNIQUENESS
VFACT(X,Y)	"The value of Y factorial is X"	TRUE	FALSE	VFACT(*,*)
C(X,Y)	"The contents of variable X is Y"	TRUE	FALSE	C(X,*)
FACT(X,Y)	"The variable X contains Y factorial"	TRUE	FALSE	FACT(X,*)
VPRODUCT(X,Y,Z)	"X is equal to the product of Y and Z"	TRUE	FALSE	FALSE
INTEGER(X)	"X is an integer"	FALSE	FALSE	FALSE
ISVAR(X)	"X is a variable"	FALSE	FALSE	FALSE
NEWVAR(X)	"X is a new local variable"	TRUE	FALSE	FALSE
=(X,Y)	"X equals Y"	TRUE	FALSE	FALSE

AXIOM	ANTECEDENT	CONSEQUENCE
TAFACT	{=(V9,1)∧=(V1Ø,1)} ∨ VFACT((DIV V9 V1Ø),(SUB1 V1Ø));	VFACT(V9,V1Ø);
TAPROD	{=(V5,Ø)∧=(V6,Ø)} ∨ VPRODUCT((MINUS V5,V3),(SUB1 V6),V3);	VPRODUCT(V5,V6,V3);

SIMPLIFICATION RULES

(ADD1(SUB1 X)) → X
(SUB1(ADD1 X)) → X
(MINUS(PLUS X Y)Y) → X
(DIV(PROD X Y)Y) → X

FUNCTION OUTPUT SYNTAX

(ADD1 X) = (X + 1)
(SUB1 X) = (X - 1)
(PLUS X Y) = (X + Y)

Figure 10

ITERATIVE RULES

RULE NAME	TFACT	TPROD
BASIS CONDITION	NEWVAR(V7)∧INTEGER(V4) ∧VFACT(V5,V6)∧C(V3,V5) ∧C(V7,V6);	NEWVAR(V4)∧C(V4,Ø) ∧C(V1,Ø);
INVARIANT	C(V7,V1Ø)∧C(V3,V9) ∧VFACT(V9,V1Ø);	C(V4,V6)∧C(V1,V5) ∧VPRODUCT(V5,V6,V3);
ITERATION STEP	C(V7,(ADD1 V1Ø))∧ PRODUCT(V3,V4,(ADD1 V1Ø));	C(V4,(ADD1 V6)) C(V1,(PLUS V2,V3));
GOAL	FACT(V3,V4);	PRODUCT(V1,V2,V3);
TEST	¬=(V1Ø,V4);	¬=(V6,V2);
OUTPUT ASSERTION	C(V3,(FAC V4));	C(V1,(PROD V2,V3));

Figure 10 - continued

```
PROC1(XØ N)
ISVAR(XØ);INTEGER(N);
COMMENT
INPUT ASSERTIONS:
NONE
OUTPUT ASSERTIONS:
C(XØ (FAC N));
   BEGIN
p(P)(TFACT)——————————  XØ ← 1;
Initial Assignment————  Y4 ← 1;
    (TFACT)
                       WHILE ¬= (Y4 N) DO
                          BEGIN
p(P)(TPROD) (Optimized Out)  Y4 ← (Y4 + 1);
                          Y1 ← Ø;
Initial Assignment (TPROD)   Y2 ← Ø;
                          WHILE ¬=(Y1 XØ) DO ——— p(R)(TFACT)
                             BEGIN
p(R) (TPROD)——————————————   Y2 ← (Y2 + Y4);
                             Y1 ← (Y1 + 1);
UPDATE Assignments (TPROD)
    (Optimized Out)          END
UPDATE Assignment (TFACT)——  XØ ← Y2;
                          END
                       END
```

Figure 11

7. PROGRAMMING AIDS

The complexity of programs that can be generated using the system is increased by some simple facilities described in this section. The capabilities discussed here are incremental extension of a current program, use of a program library, and expansion of assumptions.

The system enables a user to plan incremental extensions of a program simply by saving each completed program segment A and its output state O in a stack. The user may then pose a new goal G and solve the problem O{B}G. The composition A;B will then be output. He may choose to start from any previously saved state and associated program segment.

7.1 PROGRAM LIBRARY

When a program A has been generated to solve P{A}Q, the user may request that it be "generalized" and filed in the program library where it may be accessed by the subgoaler (similar use of a library in robot planning is reported in [Fikes,Hart, and Nilsson 1972]).

Generalization is a process which constructs a procedure declaration for the library as follows. Let I and O be the input-output assertions computed for A during its construction. We assume P⊃I, O=Q∧O', and I{A}O. The non-fluent conjuncts of I are taken as the type declarations, their variables being the parameters of the new procedure. These actual parameters are replaced throughout I{A}O by new formal parameter variables. An entry of the form:

((<procname> <goal> <effects> <type conditions> <state condition>)<body>)

is made in the library, where <procname> is a name and parameter list, <goal> is Q, <effects> is O', <body> is A, and it is assumed that

<type conditions> ∧ <state condition>{<procname>}<goal> ∧ <effects>

Library procedures are used during program generation by matching on the <goal> then establishing the <type conditions> and <state conditions> as subgoals in that order. If the conditions are satisfied then the instantiated <body> is included in the program. The system requirement of achieving the input assertions and processing the output assertion during update for a program taken from the library prevent its incorrect use in a particular program. There is no attempt to organize the library for efficient selection; the system merely tries all library procedures before any frame rule.

As an example of program assembly using the library consider the task of building a tower to reach an object, i.e. achieve "HAS(M,B)". Use will be made of a library program to find and put on shoes which achieves WEARIN(M,SHOES), previously generated using the same Frame. The generated program is then extended interactively by posing a new goal, AT(M,P).

A robotics frame for this problem is shown in figure 12, and the generated programs in figure 13.

RELATIONS	DEFINITION	FLUENT	PARTIAL	UNIQUENESS
ROBOT(X)	"X is a robot"	FALSE	FALSE	FALSE
BOX(X)	"X is a box"	FALSE	FALSE	FALSE
AT(X,Y)	"X is at location Y"	TRUE	FALSE	AT(X,*)
ON(X,Y)	"X is on Y"	TRUE	FALSE	ON(X,*)
HAS(X,Y)	"X has possession of Y"	TRUE	FALSE	FALSE
STACKED(X,Y,Z)	"X is stacked on Y at location Z"	TRUE	FALSE	FALSE
INSTACK(X,Y)	"X is in a stack at location Y"	TRUE	FALSE	INSTACK(X,*)
STACKHEIGHT(X,Y)	"the stack height at location Y is X"	TRUE	FALSE	STACKHEIGHT(*,Y)
HEIGHT(X,Y)	"X is positioned at a height of Y"	TRUE	FALSE	HEIGHT(X,*)
TOP(X,Y)	"X is the top object in stack at Y"	TRUE	FALSE	TOP(*,Y)
HIENUF(X,Y,Z)	"X is as high as Y at Z"	TRUE	FALSE	FALSE
HOLDING(X,Y,Z)	"X is holding Y at location Z"	TRUE	FALSE	HOLDING(X,*,Z)
CHAIR(X)	"X is a chair"	FALSE	FALSE	FALSE
CLOTHES(X)	"X is an article of clothing"	FALSE	FALSE	FALSE
UNDER(X,Y)	"X is under Y"	TRUE	TRUE	FALSE
WEARING(X,Y)	"X is wearing clothing Y"	TRUE	FALSE	FALSE
FOUND(X,Y)	"X found Y"	TRUE	FALSE	FALSE
=(X,Y)	"X is equal to Y"	FALSE	FALSE	FALSE
ABOVER(X,Y,Z)	"object X is above robot Y at Z"	TRUE	FALSE	FALSE
ABOVE(X,Y,Z)	"object X is above object Y at Z"	TRUE	FALSE	FALSE
BOTTOMBOX(X,Y)	"X is the bottom box at Y"	TRUE	FALSE	FALSE
BOTTOMBOXU(X,Y,Z)	"X is the bottom box at Z under Y"	TRUE	FALSE	FALSE
BELOWR(X,Y,Z)	"object X is below robot Y at Z"	TRUE	FALSE	FALSE
BELOW(X,Y,Z)	"object X is below object Y at Z"	TRUE	FALSE	FALSE
SUPPLY(X)	"the supply is at location X"	FALSE	FALSE	FALSE
NEXTBOX(X,Y)	"X is the next box after Y"	TRUE	FALSE	FALSE

Figure 12

PRIMITIVE PROCEDURE	PRE-CONDITIONS	POST-CONDITIONS
travel (R1,L1,L2) "R1 travels from L1 to L2"	ROBOT(R1)∧AT(R1,L1)∧HEIGHT(R1,0);	AT(R1,L2);
move(R1,O1,L1,L2) "R1 moves O1 from L1 to L2"	ROBOT(R1)∧BOX(O1)∧AT(O1,L1)∧¬ INSTACK(O1,L1)∧ CLOTHES(O3)∧WEARING(R1,O3)∧AT(R1,L1);	AT(O1,L2)∧AT(R1,L2);
stack(R1,O2,O1,L1) "R1 stacks O2 on O1 at L1"	ROBOT(R1)∧BOX(O1)∧BOX(O2)∧≠(O1,O2)∧AT(O1,L1)∧ AT(O2,L1)∧AT(R1,L1)∧HOLDING(R1,O2,L1)∧ HEIGHT(R1,0)∧ON(R1,O1,L1)∧¬STACKED(O3,O1,L1) ∧STACKHEIGHT(H1,L1);	STACKED(O2,O1,L1)∧ STACKHEIGHT((EVN(ADD1 H1)),L1) ∧TOP(O1,L1);
climb(R1,O1,L1) "R1 climbs O1 at L1"	ROBOT(R1)∧ABOVER(O1,R1,L1)∧AT(R1,L1)∧ ¬INSTACK(O1,L1)∨ {STACKED(O1,O2,L1)∧ON(R1,O2,L1)}∧ REQUEST(HEIGHT(R1,H1));	ON(R1,O1,L1)∧ HEIGHT(R1,(EVN(ADD1 H1)));
unclimb(R1,O2,L1) "R1 unclimbs O2 at L1"	ROBOT(R1)∧BELOWR(O1,R1,L1)∧AT(R1,L1)∧ REQUEST(HEIGHT(R1,H1))∧ REQUEST(STACKED(O2,O1,L1))∧ ON(R1,O2,L1);	ON(R1,O1,L1)∧ HEIGHT(R1,(EVN(SUB1 H1)));
stepoff(R1,O1,L1) "R1 steps off O1 at L1"	=(H1,0)∧HEIGHT(R1,1)∧ON(R1,O1,L1);	HEIGHT(R1,H1)∧ ¬ON(R1,O1,L1);
reach(R1,O1,L1) "R1 reaches O1 at L1"	ROBOT(R1)∧AT(O1,L1)∧HIENUF(R1,O1,L1);	HAS(R1,O1);
lift(R1,O1,L1) "R1 lifts O1 at L1"	ROBOT(R1)∧BOX(O1)∧AT(O1,L1)∧AT(R1,L1)∧ ¬INSTACK(O1,L1);	HOLDING(R1,O1,L1);
find(R1,O1,L1) "R1 finds O1 at L1"	ROBOT(R1)∧CHAIR(O2)∧AT(O2,L1)∧AT(R1,L1)∧ UNDER(O1,O2);	FOUND(R1,O1);
put_on(R1,O1) "R1 puts on O1"	ROBOT(R1)∧CLOTHES(O1)∧FOUND(R1,O1);	WEARING(R1,O1);

AXIOM	ANTECEDENT	CONSEQUENCE
TABOVER	¬ON(R1,O2,L1)∨{ON(R1,O3,L1)∧ABOVE(O1,O3,L1)};	ABOVER(O1,R1,L1);
TABOVE	=(O1,O3)∨{STACKED(O2,O3,L1)∧ABOVE(O1,O2,L1)};	ABOVE(O1,O3,L1);
TBELWR	ON(R1,O2,L1)∧BELOW(O1,O2,L1);	BELOWR(O1,R1,L1);
TBELOW	=(O1,O3)∨{STACKED(O3,O2,L1)∧BELOW(O1,O2,L1)};	BELOW(O1,O3,L1);
TBOT	TOP(O3,L1)∧BOTTOMBOXU(O1,O3,L1);	BOTTOMBOX(O1,L1);
TBOTU	STACKED(O3,O4,L1)∧STACKED(O4,O2,L1)∨ STACKED(O3,O1,L1)∧¬STACKED(O4,O2,L1)∨ BOTTOMBOXU(O1,O4,L1);	BOTTOMBOXU(O1,O3,L1);
TNEXT	SUPPLY(L1)∧AT(O4,L1);	NEXTBOX(O4,O3);
TINSTACK	TOP(O2,L1)∧BELOW(O1,O2,L1);	INSTACK(O1,L1);

DEFINITION

TWITE HEIGHT(O1,H1)∧STACKHEIGHT(H1,L1)∧TOP(O2,L1)∧ON(R1,O2,L1) ≡ HIENUF(R1,O1,L1)

Figure 12 - continued

ITERATIVE RULE	BASIS CONDITION	INVARIANT	ITERATION STEP	GOAL	TEST	OUTPUT ASSERTION
TUP	REQUEST(HEIGHT(R1,H2)) ∧GZ(H2)∨ (BOTTOMBOX(O3,L1) ∧ON(R1,O3,L1));	ON(R1,O1,L1)∧ STACKED(O2,O1,L1) ∨TOP(O1,L1);	ON(R1,O2,L1);	HEIGHT(R1,H1);	--	--
TDOWN	GZ(H1)∧ REQUEST(HEIGHT(R1,H2)) ∧GT(H2,H1);	ON(R1,O1,L1)∧ STACKED(O1,O2,L1) ∨BOTTOMBOX(O1,L1);	ON(R1,O2,L1);	HEIGHT(R1,H1);	--	--
TSTA	STACKED(O2,O1,L1) ∧ON(R1,O2,L1);	TOP(O3,L1)∧ STACKHEIGHT (H2,L1)∧ NEXTBOX(O4,O3);	HOLDING(R1,O4,L1) ∧HEIGHT(R1,H2) ∧STACKED(O4,O3,L1);	STACKHEIGHT (H1,L1);	--	--

INITIAL STATE

ROBOT(M)∧BOX(B2)∧BOX(B5)∧BOX(B3)∧BOX(B6)∧BOX(B4)∧BOX(B7)∧AT(M,P)∧AT(B,U)∧AT(B2,SLOC)∧AT(B5,SLOC)∧AT(B3,SLOC)∧
AT(B6,SLOC)∧AT(B4,SLOC)∧AT(B7,SLOC)∧SUPPLY(SLOC)∧STACKHEIGHT(Ø,U)∧HEIGHT(M,Ø)∧HEIGHT(B,4)∧CLOTHES(SHOES)∧
CHAIR(CHAIR1)∧CHAIR(CHAIR2)∧AT(SHOES,CORNER)∧AT(CHAIR1,CORNER)∧AT(CHAIR2,CORNER);

ADVICE

RECURSIVE RULES: CLIMB,TABOVE,TBELOW,TBOTU

PAIRWISE INEQUALITIES: travel(R1,*,*),move(R1,O1,*,*)
STACK(R1,*,*,L1)

Figure 12 - continued

```
PROC1(M SHOES)
ROBOT M);CHAIR CHAIR?);CLOTHES SHOES);
COMMENT
INPUT ASSERTION:
HEIGHT M  Ø )∧AT(M P)∧AT(CHAIR? CORNER)
OUTPUT_ASSERTION:
AT(M CORNER)∧FOUND M SHOES)∧WEARING(M SHOES);
COMMENT
PROC ATTEMPTS_TO_ACHIEVE_ (FOUND M SHOES);
   BEGIN
   TRAVEL(M P CORNER);
   IF ¬UNDER(SHOES CHAIR?) THEN
      PROC2(M SHOES)
   ELSE
      BEGIN
      FIND(M SHOES CORNER)
      END
   PUT_ON(M SHOES)
   END
```

```
PROC3(M B)
ROBOT(M);BOX(B );CLOTHES(SHOES);CHAIR(CHAIR?);BOX(B4);SUPPLY(SLOC);BOX(B6);BOX(B3);
COMMENT
INPUT_ASSERTION:
AT (M P)∧AT(B7 SLOC)∧HEIGHT(M Ø )∧AT(CHAIR? CORNER)∧AT(B4 SLOC)
∧HEIGHT(B 4)∧STACKHEIGHT(Ø U)∧AT(B6 SLOC)∧AT(B3 SLOC)
OUTPUT_ASSERTION:
AT(M P)∧AT(B7 U)∧AT(B4 U)∧STACKED(B4 B7 U)∧AT(B6 U)
∧STACKED(B6 B4 U)∧STACKHEIGHT(4 U)∧HAS(M B)∧HEIGHT(M Ø)
∧FOUND(M,SHOES)∧WEARING(M,SHOES);∧AT(B3 U)∧STACKED(B3 B6 U);
   BEGIN
   TRAVEL(M P CORNER);
   IF¬UNDER(SHOES CHAIR?) THEN
      PROC2(M SHOES)
   ELSE
      BEGIN
      FIND(M SHOES CORNER)
      END
   PUT_ON(M SHOES);
   TRAVEL(M CORNER SLOC);
   MOVE(M B7 SLOC U);
   TRAVEL(M U SLOC);
   MOVE(M B4 SLOC U);
   LIFT(M B4 U);
   CLIMB(M B7 U);
   STACK(M B4 B7 U);
   CLIMB(M B4 U);
   Y3 ← 2;
   Y4 ← B4;
   IF NEXTBOX(W4 Y4) THEN
      Z4 ← W4;
   WHILE ¬STACKHEIGHT(4 U) DO
      BEGIN
      Z3 ← ADD1(Y3);
      Y1 ← Y4;
      IF STACKED(Y1 W1 U) THEN
         Z1 ← W1;
      WHILE ¬HEIGHT(M I)DO
         BEGIN
         UNCLIMB(M Y1 U);
         Y1 ← Z1;
         IF STACKED(Y1 WI U) THEN
            Z1 ← WI;
         END
      STEPOFF(M B7 U);
      TRAVEL(M U SLOC);
      MOVE(M Z4 SLOC U);
```

Assembled from Library →

Figure 13

```
            LIFT(M Z4 U);
            CLIMB(M B7 U);
            Y2 ← B7;
            IF STACKED(W2 Y2 U) THEN
               Z2 ← W2;
            WHILE ¬HEIGHT(M Y3) DO
               BEGIN
               CLIMB(M Z2 U);
               Y2 ← Z2;
               IF STACKED(W2 Y2 U) THEN
                  Z2 ← W2;
               END
            STACK(M Z4 Y4 U);
            Y3 ← Z3;
            Y4 ← Z4;
            IF NEXTBOX(W4 Y4) THEN
               Z4 ← W4;
            END
         CLIMB(M B3 U);
         REACH(M B U);
         Y5 ← B3;
         IF STACKED(Y5 W5 U) THEN
            Z5 ← W5;
         WHILE ¬HEIGHT(M 1 U) DO
            BEGIN
            UNCLIMB(M Y5 U);
            Y5 ← Z5;
            IF STACKED(Y5 W5 U) THEN
               Z5 ← W5;
            END
         STEPOFF(M B7 U);
         TRAVEL(M U P);
         END
```

Incremental Extension →

Figure 13 - continued

7.2 EXPANSION OF ASSUMPTIONS

A basic capability for structuring programs is provided by interactively allowing the user at any level in program generation to define a primitive procedure, P{p}Q, as an assumption. The program generator will then use p as usual except at each point of call to p in the program the current state I' and current goal G will be saved. The triple <p,I',G> is placed in a stack of subtasks for later expansion.

When a program containing assumed primitive procedures has been generated, the user is given the list of assumptions his program depends on and allowed to selectively expand them in terms of lower level procedures. For the subtask <p,I',G>, the state is initialized to I', the frame may be changed, G is given as the goal,and a body for the procedure p is generated.

Consider the example given in Section 6 of computing the value of n factorial where multiplication is not a primitive operation. The initial frame is the same except that in place of an iterative rule for multiplication, there is an assumed primitive procedure

ISVAR(V1){times(V1,V2,V3)}PRODUCT(V1,V2,V3),

where PRODUCT(V1,V2,V3)≡C(V1,(PROD V2,V3)).

The program generated using this frame is given in figure 14. To expand the non-primitive procedure "times(V1,V2,V3)" the full frame including the iterative product rule is given and the sub-program generated is shown in figure 15.

In the current implementation it is assumed tht the expanded sub-programs will have no side effects. However this assumption could be removed by a mechanism similar to checking rejoin conditions for contingency programs (Section 5.4).

To develop a useful structured programming system interaction appears essential along with further study about how humans do (or should do) programming.

```
PROC1(X∅ N)
ISVAR(X∅);INTEGER(N);
COMMENT
INPUT ASSERTION:
NONE
OUTPUT ASSERTION:
C(X∅ (FAC N));
COMMENT
THIS PROGRAM RELIES ON THE FOLLOWING ASSUMPTIONS:
(TIMES)
   BEGIN
   X∅ ← 1;
   Y1 ← 1;
   WHILE ¬ >(Y1 N) DO
      BEGIN
      Y1 ← Y1+1;
      TIMES(X∅ X∅ Y1)
      END
   END
```

Figure 14

```
TIMES(X∅ Y1 Z1)
ISVAR(X∅);
COMMENT
INPUT ASSERTION:
NONE
OUTPUT ASSERTION:
C(X∅ (PROD Y1 Z1));
   BEGIN
   X∅ ← ∅;
   Y∅ ← ∅;
   WHILE ¬ =(Y∅ Y1) DO
      BEGIN
      Y∅ ← Y∅+1;
      X∅ ← X∅+Z1;
      END
   END
```

Figure 15

8. CORRECTNESS OF THE FORMAL ALGORITHM

The basic problem solving algorithm implemented in the system is that of problem reduction subgoaling with backtrack. In this section the formal algorithm will be given and a proof of its correctness sketched.

8.1 BACKTRACK PROGRAMMING

Backtrack programming describes an exhaustive search procedure appropriate for solving problems of the form:

Given a collection of sets $X_1,X_2,...,X_n$ (goals to be achieved), select a sequence of elements $(x_1,x_2,...,x_n)$, one from each set (a way of achieving each goal), such that some criterion function $f(x_1,x_2,...,x_n)$ is maximized. In general f may be numerical valued or simply have the values of either "success" or "failure" for any sequence $(x_1,x_2,...,x_k)$, $k \leq n$, and it is possible to determine whether or not a partial solution is inherently suboptimal, i.e. if there does not exist a successful $x_k \in X_k$ given the current choice of $x_1,...,x_{k-1}$.

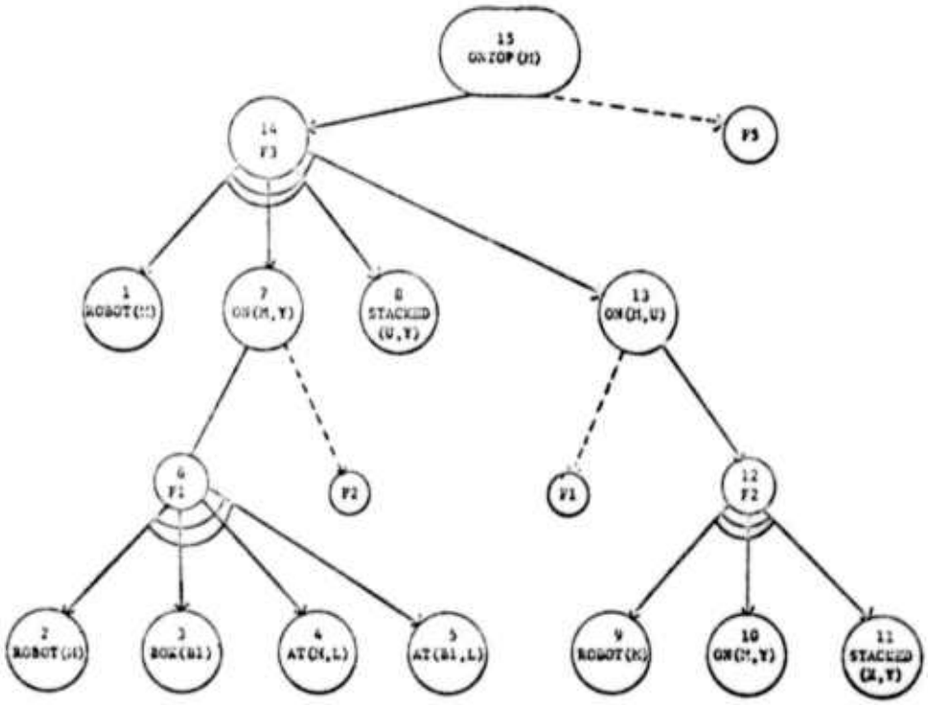
To control such a search a program must have the abilities to enumerate the alternatives for selection at the kth level, e.g., $x_{k1},x_{k2},...x_{km}$ (enumerate function), select one, say x_{k1} (choose function), and repeat this process at successively higher levels, i.e., k+1, k+2,..., until either the nth level is attained or a partial solution $(x_1,x_2,...,x_k,...x_p)$, $p \leq n$, is reached that is inherently suboptimal, i.e., no selection can be made at level p that is correct with respect to the previous choices already made. In the latter case the program must "backtrack" to a previous level, e.g., k, at which a "new" selection (different from previous level k selection) can be made that achieves a correct kth level solution (unchoose function). The process then continues in the "forward" direction. Ultimately either an nth level sequence is found that is satisfactory or the

operation of the program has proven that a solution does not exist, i.e., the program has "backtracked" to the 0th level and has failed to solve the problem.

8.2 TRAVERSING THAND-OR-AND SUBGOAL TREES

Programs are generated by using rules and axioms to prove that the output program transforms the initial state into one in which the given goal condition is true. Frame rules act as partial functions on the domain of possible states, defined only on those states in which their premisses are true and transforming them into states in which their postconditions (or goal conditions) are true.

In figure 16 is given the subgoal tree traversed during the solution of the example problem given in Section 2. Goal nodes are labeled with the goal and an integer indicating the order of achievement in the depth-first search. Rule nodes (used to expand the goals) are labeled with the rule name and an integer indicating the order of successful application. In the tree absence of angle marks indicate OR connection, a single angle mark indicates AND connection and double angle marks indicate THAND connection.



PROBLEM 1: THAND-OR-AND TREE SEARCH
Figure 16

Program generation is done by computing on a triple <G',I',A>, where G' is the subgoal to be attempted next, I' is the current state and A is the current program segment. For each rule used, an instantiation of the associated program construct, if any, is added to A using rule R2. The general form of rules to expand goals (as explained in Section 2.1) is,

$$\frac{H_1, \ldots, H_n}{K}$$

The instantiation of program constructs is built up in a substitution α that replaces variables in the frame rules by terms from the initial state. For any rule if $K\alpha = G'$ then that rule is applicable to the achievement of G' and the premisses, $H_1\alpha, \ldots, H_n\alpha$ are the subgoals whose solution implies G'. We assume for the computation of α that variables in different applications of the same rule are distinct.

The syntax of assertions used in rules, axioms, definitions and state descriptions is given in Section 3.1. Consider the restrictions that the exclusive or "⊕" is used only as a top level connective in disjunctive postconditions of primitive procedures and the thand "&" is only used to connect the premisses of an iterative rule (which in fact follows the current implementation though its effect can be gained in any rule using advice). Then for any <goal node>, say G' in state I', the THAND-OR-AND solution tree Tr that may be rooted at G' is described by the following grammar Gr:

<goal node> ::= <true goal>|<prim proc>|<def>|<it rule>|<undetermined goal>
<prim proc> ::= <assertion>
<def> ::= <assertion>
<it rule> ::= (<basis> ∧ <invariant>) & <it step>
<basis> ::= <assertion>
<invariant> ::= <assertion>
<it step> ::= <assertion>
<assertion> ::= <disjunction>|<disjunction> ∧ <assertion>
<disjunction> ::= <goal node>|<goal node> ∨ <disjunction>

where if G' is a <true goal> then $(\exists\alpha)[\Gamma' \vdash F \subset G'\alpha$ and <undetermined goal> is as defined in Section 5. A full specification of the formal algorithm for processing undetermined goals would include a formalization of the subproblem stack, the methods for choosing contingency goals, assembly of conditional statements, keeping track of the goals in the scope of a contingency goal and contingency state manipulation. However since the concepts involved are described quite completely in Sections 5 and 9 they will not be dealt with further here.

The definition of an achieved goal node G' in a THAND-OR-AND tree is:

(1) If G' is a <true goal> then it is achieved,

(2) If G' has OR subgoals then it is achieved iff at least one of its subgoals is achieved,

(3) If G' has AND subgoals then it is achieved iff all of its subgoals are achieved and remain true in the resulting state.

(4) If G' has THAND subgoals then it is achieved iff all of its subgoals have been achieved.

Further details on these kinds of problems may be found in [Nilsson 1971].

If G' is achieved under (2), (3), or (4) (i.e. by rule application), then Γ' is updated by $Inv(K\alpha,\Gamma')$ and a procedure call or a while loop may be appended to A.

Search algorithms of this type may be conveniently implemented using any of the new languages that directly support subgoal tree generation, backtrack, and a data base [Hewitt 1971, Sussman and Winograd 1972], [Rulifson et al. 1972].

8.3 LABELED, ORDERED SUBGOAL TREES

Before we can consider correctness, the notion of a labeled, ordered THAND-OR-AND subgoal tree, say Tr, must be formalized. Let Tr be a solution tree generated by the algorithm during a successful program generation, S be the set of nodes in Tr, and $R \supset SXS$ be a partial ordering on S. Let J be another relation on S defined in terms of R by:

$$xJy \text{ iff } (\forall x,y)[xRy \wedge \neg yRx \wedge (\forall z)[z \neq y \wedge zRy \supset zRx]].$$

For $x,y \in S$, xJy means that y is the R-direct descendent of x, or x is the R-direct ancestor of y.

DEFINITION A structure $Tr = \langle S,R \rangle$ is a tree if the following properties are satisfied:

(1) There is a root element of the tree, i.e., $(\exists x),(\forall y)[y \in S \supset xRy]$,

(2) For $x,y,z \in S$, if xJz and yJz then $x = y$

DEFINITION A structure $Tr = \langle S,R,L \rangle$ is an ordered tree if the following properties are satisfied:

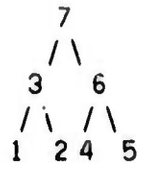
(1) $Tr = \langle S,R \rangle$ is a tree,

(2) For each $x \in S$, L is a total ordering of $\{y : xJy\}$,

(3) For each $x,y,z \in S$, if xLy and yRz and $x \neq y$ then xLz,

(4) For each $x,y,z \in S$, if xLy and xRz and $x \neq y$ then zLy.

The relation L orders the nodes of Tr in depth-first achievement order, e.g.,



Let V be the set of goals achieved in Tr instantiated by α. The function f will be called the labeling function.

DEFINITION A structure Tr = <S,V,R,L,f> is a labeled, ordered tree if the following properties are satisfied:

(1) Tr = <S,R,L> is an ordered tree,

(2) The function f maps S onto V.

Let Gr be the grammar describing solution subgoal trees and let Tr = <S,V,R,L,f> be a labeled, ordered tree.

DEFINITION Tr is a labeled, ordered THAND-OR-AND subgoal tree rooted at G' in Gr if the following properties are satisfied:

(1) If $x \in S$ is the root of the tree <S,R> then f(x) = G',

(2) If $\exists y \in S$ such that $f(y) \neq \lambda$ and xRy and $x \neq y$ then f(x) is not a <true goal>(i.e. x is not a leaf node).

(3) If $y_1,\ldots,y_n \in \{y : xJy\}$, where y_iLy_j for $i<j$ then there exist some frame rule having postcondition (or goal) f(x) and premisses $f(y_1), \ldots, f(y_n)$.

We will refer to such trees as solution trees in the next section.

8.4 CORRECTNESS

For any output program generated by the system the associated solution (sequence of axioms and rules used) provides a proof within the logic of programs given in Section 2 that the program satisfies the given input-output assertions. Because of implementation limitations, heuristic system methods, and consistency requirements in a frame definition which the user may violate a system generated program may in fact be incorrect. However we will show that from a solution tree Tr generated by the formal algorithm to solve the problem <I,G> with properties as defined, a correctness proof of the solution can be given. Conditions for correctness of the procedure for generating conditional statements was given in Section 5.

We may show by induction on the ordering of nodes in Tr that the output program A solves the problem, i.e. L(F) ||-I{A}G by showing it to be true for each subgoal and partial program, i.e. if x∈S is the root of the tree and f(x) = G then for any y∈S such that xRy , L(F) ||-I{A'}f(y), where A' is the partial program in the triple computed by the achievement of f(y).

Let G' = f(x) be such that ∀y∈S xLy, then G'∈V is a <true goal>, i.e. it labels the leftmost leaf node of Tr, and L(F) ||-I⊃G'.

As an induction hypothesis assume that for an arbitrary G'=f(y) such that y is not the root of Tr that L(F) ||-I{A'}f(y). We will then show that this must imply L(F) ||-I{A"}f(z), where yLz and either zJy or (∃x)[xJy ∧ xJz ∧ ¬ zJy], where A' and A" are the generated program segments in the associated triples.

Consider the cases for the triple <G',I',A'>,

(1) If G' labels a leaf node of Tr then L(F) ||-I' ∪ F ⊃ G' and the state and program segment are unchanged by its achievement. This implies L(F) ||-I{A'}G'.

(2) If G' labels a non-leaf node x of Tr then we have the following subcases,

An instance of a primitive procedure rule $P\alpha\{p\alpha\}Q\alpha$ is applied to achieve G', i.e. $Q\alpha \supset G'$. Its application is justified by the change of variables rule R4. By hypothesis $I' \cup F \vdash P\alpha$ since property (3) of the definition of a solution tree is satisfied. The rule of consequence implies $L(F) \Vdash I'\{p\alpha\}Q\alpha$ and invariance implies $L(F) \Vdash I'\{p\alpha\}I''$, where $I'' = Inv(Q\alpha, I')$. By the rule of composition R3 we may compose A' with $p\alpha$ and by the induction hypothesis we conclude that $L(F) \Vdash I\{A';p\alpha\}I''$, where $I'' \supset G'$.

For the cases in which instances of definitions or iterative rules are applied to achieve G', the induction step may also be proved establishing $L(F) \Vdash I\{A\}G$. More formal details may be found in [Buchanan and Luckham 1974]. This discussion was intended to justify the formal methods implemented in the system by showing that under certain assumptions about solution trees a correctness proof can be given.

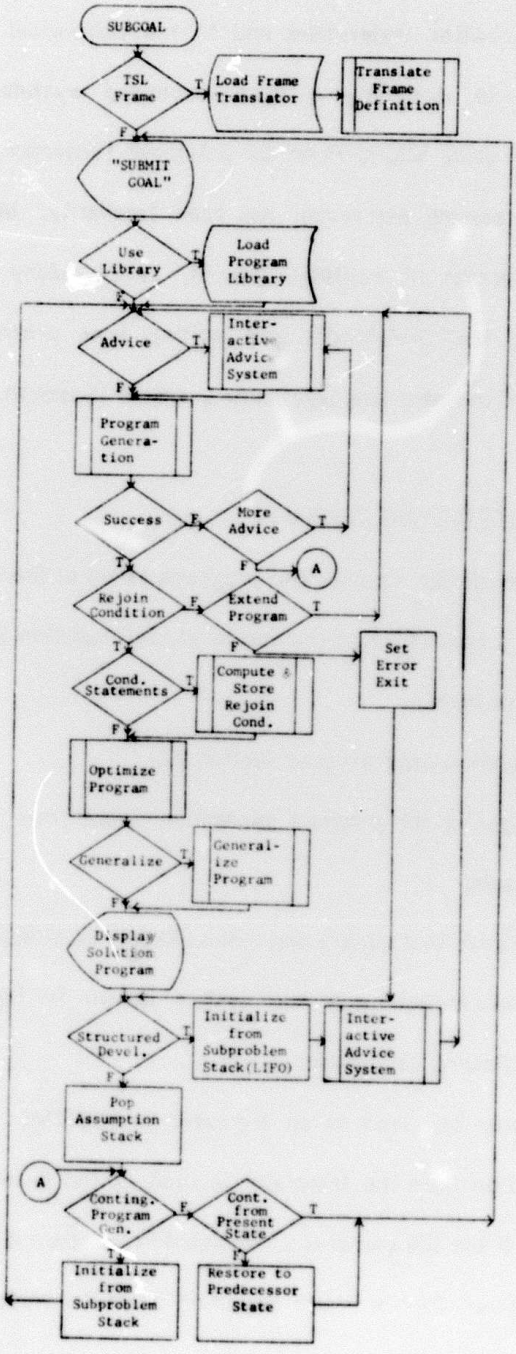

SUBGOAL
TSL Frame
T
Load Frame Translator
Translate Frame Definition
F
"SUBMIT GOAL"
Use Library
T
Load Program Library
F
Advice
T
Inter-active Advice System
F
Program Genera-tion
Success
F
More Advice
T
F
A
Rejoin Condition
F
Extend Program
T
T
F
Set Error Exit
Cond. Statements
T
Compute & Store Rejoin Cond.
F
Optimize Program
Generalize
T
General-ize Program
F
Display Solution Program
Structured Devel.
T
Initialize from Subproblem Stack(LIFO)
Inter-active Advice System
F
Pop Assumption Stack
A
Conting. Program Gen.
F
Cont. from Present State
T
T
F
Initialize from Subproblem Stack
Restore to Predecessor State

Figure 17

9. SYSTEM DESCRIPTION

This section will document the system implementation to the end that its operation might be better understood and to the conceptual level that would be reasonably helpful in designing a more expanded system. The system was implemented in LISP using MICRO-PLANNER primitives [Sussman and Winograd, 1971], with which we will assume the reader has some familiarity. MICRO-PLANNER was a very preliminary version of PLANNER [Hewitt, 1972]. Many of our programming triumphs modifying MICRO-PLANNER and writing new primitives are no longer necessary in light of the new languages now available [Sussman, 1972],[Rulifson et al. 1972].

9.1 OVERVIEW OF INTERACTIVE SYSTEM USE

The interactive decision points and programs called at the top level are shown in figure 17. (This is a flow chart of the top level LISP function SUBGOAL.) The system basically has three segments:

(1) a Frame translation program (see Section 9.2),

(2) a set of programs for program generation called from SUBGOAL and using a translated Frame,

(3) a set of primitives that modify and extend MICRO-PLANNER.

The user's interaction with the system shown in figure 17 is informally described by the following procedure:

(1) A filename may be given as an argument to SUBGOAL. If the file contains a Frame definition then the translator is read in, the Frame definition translated and loaded. If the file contains a translated Frame then it is simply loaded. If no filename is given then a Frame definition to be translated and loaded is given interactively from the terminal.

(2) When prompted by the system, the user inputs a goal to be achieved by the program to be generated.

(3) If the user desires to give advice to the system relative to the given goal then the advice system is called.

(4) The subgoaling system uses the translated Frame to attempt to generate a program achieving the goal. If unsuccessful the user may try again with more advice (go to (3)).

(5) If rejoin conditions (see Sections 5.4 and 9.4) are pending for this generated procedure then they are tested for satisfaction. If they are not satisfied then the user may attempt to extend the procedure to yield a state in which they are true. If he chooses not to do this or the system fails in its attempt then an error exit is substituted for any call to that procedure in its trunk program.

(6) If the user elects, the program may be optimized according to some simple criteria, e.g. elimination of dead assignment statements and reduction of the number of program variables.

(7) The user may then choose to have the generated program generalized and filed in a program library.

(8) The program is then displayed for visual inspection.

(9) If there are conditional statements (see Section 9.5) then the user may elect to do contingency program generation. If so then the state, goal pending and answer is initialized from the stack of contingency tasks (go to (3)).

(10) If any assumed primitive procedures occur in the generated program the user is informed and may structurally (see Section 9.7) develop each assumption procedure call by generating a program whose input and output assertions

match the pre and postconditions of the assumed primitive procedure (Initialize the state and go to (3)).

(11) The program may now either be incrementally extended from the current state by giving an additional goal (go to (2)) or any previously completed program segment and final state may be returned to and extended.

In Appendix B is an example of an interactive dialogue including a Frame translation and program generation.

9.2 PROCEDURAL REPRESENTATION OF A FRAME

In Section 8 the basic problem reduction subgoaling algorithm was given and in Section 4 associated problem solving processes using Frame information were described. In this section a more detailed description of the function and form of a translated Frame will be given. The transiation and use of iterative rules and the generation of conditional statements will be given in Sections 9.6 and 9.5 respectively.

9.2.1 SOME ELEMENTS OF MICRO-PLANNER

Assuming general familiarity with MICRO-PLANNER we will briefly describe a few basic primitives and theorem types as used in the system description (see [Sussman and Winograd, 1971] or [Baumgart, 1972]). In the current implementation, <pattern> will represent some relation. In a more general treatment <pattern> could represent an arbitrary Boolean expression of relations. Pattern matching is restricted to simple unification.

The control structure is a backtrack stack interpreter consisting of a program representation, a processor state, a processor and a push down stack of all previous states the processor has been through since the beginning of a particular computation. The processor may backtrack to a previous state and exhaustively search a subgoal tree in a depth-first manner trying all possible variable bindings and applicable theorems to return success ultimately to the top goal node.

(1) (THGOAL <pattern> <recommendation>). If no <recommendation> is given then THGOAL simply searches the data base for an assertion that matches the pattern. If it finds one, it succeeds and carries out the unification substitution for any variables in the pattern, otherwise it fails. If a <recommendation> is given it will be of the form, (THTBF <filter>), where <filter> is the name of a

unary LISP function that selects from candidate THCONSE theorems to be called after a data base search has failed. The atom THTRUE is the always true filter, i.e. allows any matching theorem to be tried.

(2) (THNOT <argument>). THNOT fails if its <argument> succeeds, otherwise it fails.

(3) (THASSERT <skeleton> <recommendation>). The <skeleton> may either be a theorem name to be put on a ready-to-use list or an instantiated relation that is to be added to the data base. THASSERT fails only if it tries to assert a <skeleton> already existing either on the ready-to-use list or in the data base. If a <recommendation> is given it will be of the form, (THTBF <filter>), where <filter> is the name of a unary LISP function that selects from candidate THANTE theorems to be called.

(4) (THERASE <skeleton> <recommendation>). The <skeleton> may either be a theorem name to be removed from a ready-to-use list or an instantiated relation that is to be removed from the data base. THERASE fails only if it tries to remove a <skeleton> that does not exist either on the ready-to-use list or in the data base. If a <recommendation> is given it will be of the form, (THTBF <filter>), where <filter> is the name of a unary LISP function that selects from candidate THERASING theorems to be called.

(5) (THOR <arg1>...<argn>. THOR succeeds if one of its arguments succeeds where evaluation is left to right. This construct is used to implement logical disjunction.

(6) (THSETQ $\langle var_1 \rangle$ $\langle e_1 \rangle$ ... $\langle var_n \rangle$ $\langle e_n \rangle$). This primitive assigns the value of expression $\langle e_i \rangle$ to the variable $\langle var_i \rangle$ for $i=1,...,n$. The variable is not evaluated. This assignment is undone if failure backs up to it. A simple extension provided the function THSET which does evaluate its first argument but is otherwise equivalent to THSETQ.

(7) Theorems. Theorems are pattern evoked and have the format:

(DEFPROP <name>(<type> <varlist> <pattern> <body>) THEOREM)

where
<name> is an atomic theorem name,
<type> is the theorem type,
<varlist> is the list of variables used,
<pattern> is a relation for pattern match invocation,
<body> is a sequence of statements having the syntax of the body of a LISP PROG.

The list (<type> <varlist> <pattern> <body>) is of course attached to the property list of <name> under the indicator THEOREM as a result of the DEFPROP.

There are three types of theorems:

(a) Consequent theorems (THCONSE) are called by a match between the pattern of a THGOAL statement and the theorem pattern.

(b) Erasing theorems (THERASING) are called by a match between a relation skeleton of a THERASE statement and the theorem pattern.

(c) Antecedent theorems (THANTE) are called by a match between a relation skeleton of a THASSERT statement and the theorem pattern.

If a theorem's pattern is matched by the appropriate calling statement then the <varlist> is bound and body is executed such that for the theorem to succeed each statement must succeed (return non-nil) with the capability for backtracking to discover an instantiation and/or subgoal tree rooted at that statement that returns successfully. THERASING and THANTE theorems do not return a value and are assumed to succeed as it affects the success of the calling statement; however, THCONSE theorems return either a success or a failure value that determines the success of the calling statement.

9.2.2 SPECIFICATION AND BASIC FUNCTIONS OF FRAME RULES

When a primitive procedure is initially input the following information is put on the property list of the atomic procedure name:

(1) preconditions,
(2) postconditions,
(3) argument list,
(4) whether or not the procedure is an assumption,
(5) whether or not the procedure is recursive,
(6) inequalities desired in argument positions,
(7) indicator of rule type, i.e. primitive procedure.

Except for argument list specifications ((3) and (6)), the analogous information for axioms and definitions is initially stored the same way.

On the property list of each predicate symbol is stored the following:

(1) argument list,
(2) whether or not the relation is a fluent,
(3) whether or not the relation is partial,
(4) argument positions having the uniqueness property.

The internal data structure used to represent assertions after input is a list of lists where the interpretation of juxtaposition of elaments is conjunction at the top level and alternates between conjunction and disjunction at successive levels of nesting. At the bottom level a literal is represented as a list of negation sign (if any), predicate symbol and the arguments. For example the assertion,

$P(X) \vee Q(Y) \wedge \neg R(X,Y) \wedge S(Z,X) \vee \{T(Z) \wedge M(V)\}$;

is represented as

((P X) (Q Y)) ((¬ R X Y)(S Z X)((T Z))((M V))))).

This internal representation is clearly adequate for assertions input using the syntax given in Section 3.

A translated rule for a primitive procedure contains the basic functional segments shown in figure 18. An actual example of a primitive procedure definition and its translated form is given in Section 9.2.3.1. The pattern is the postcondition

achieved by an application of the procedure. The interactive program allows the user to enter the Advice System then return for continuation of subgoaling. Trace information of current path and goals pending is displayed. Nonfluent preconditions are achieved first then the fluent preconditions. The mechanism for achieving a conjunction of fluent goals is described in Section 9.2.3.2. Processing to make the state consistent with the postcondition next to be asserted is carried out. The instantiated procedure call is appended to the partially generated program followed by processing for collecting input-output assertions, forming correct block structure in nested conditional statements and diagnostic output to the user.

CALLING PATTERN

INTERACTIVE AND TRACE PROGRAMS

ACHIEVEMENT OF NON-FLUENT PRECONDITIONS

ACHIEVEMENT OF FLUENT PRECONDITIONS

STATE CONSISTENCY PROCESSING

ASSERTION OF POSTCONDITIONS

ASSEMBLY OF PROCEDURE CALL INTO PROGRAM

MISCELLANEOUS PROCESSING
(INPUT-OUTPUT ASSERTION, BLOCK STRUCTURE, DIAGNOSTICS)

FUNCTIONAL SEGMENTS OF RULES FOR PRIMITIVE PROCEDURES
FIGURE 18

9.2.3 FRAME TRANSLATION

A Frame defined using the language described in Section 3 is translated into a set of LISP functions and MICRO-PLANNER theorems that form the basis for the subgoaler. In particular for each rule or axiom, one or more MICRO-PLANNER THCONSE theorems is constructed, for each distinct predicate symbol a THERASING theorem is generated to implement the conjunction connective. The initial state description is converted to assertions placed in the data base.

9.2.3.1 TRANSLATION PROCEDURE

The translation is carried out according to the following procedure:

(1) The appropriate input device, i.e. terminal or disk, from which to read the Frame definition is selected.

(2) For each rule defined the information listed in Section 9.2.2 is input and internalized.

(3) The initial state expressed in the syntax of conditions is input and internalized.

(4) For each predicate symbol used, the information listed in Section 9.2.2 is input and internalized.

(5) The user may request context linking or performance statistics to be gathered.

(6) Algebraic simplification rules may be given of the form,

t → t', where t,t' are terms,

which are used to reduce t to t' should t occur in an argument of a relation in a THGOAL statement, e.g.

(MINUS(PLUS X,Y)Y) → X,

where X and Y may be bound to arbitrary terms to which the rules will be applied recursively.

(7) Conversion rules for more readable output syntax for functions occurring in the generated program are specified in the form,

t = α , where t is a term and α is an expression in the new syntax,

which are used to produce the final output form of the generated program, e.g.

(PLUS X Y) = (X + Y),

where X and Y may be bound to arbitrary terms to which the rules will be applied recursively.

(8) The conjunction of literals given to form the initial state is asserted into the data base according to the following rules giving the assertion made for a literal l as a function of being negated or partial.

(a) If l = P(t1,...tn), for some predicate P, then assert P(t1,...,tn),

(b) If l = ¬P(t1,...,tn) and P is total then assert nothing,

(c) If l = ¬P(t1,...,tn) and P is partial then assert, by convention, NP(t1,...,tn).

(9) For each predicate symbol used generate a THERASING theorem and some global variables whose form and use in implementing conjunction are described in Section 9.2.3.2.

(10) For each rule defined, a THCONSE theorem is generated implementing the functions shown in figure 18, i.e.

(a) The calling pattern is the rule postcondition.

(b) For each total precondition literal l a THGOAL statement is generated according to the rules:

(i) If l = P(t1,...,tn) and P is non fluent then

"(THGOAL(P α(t1)...α(tn))(THTBF FILTERAX))"

where FILTERAX is a LISP filter function which permits only deduction

using the axioms relative to the current state and α transforms ti into a valid MICRO-PLANNER term.

(ii) If l = ¬P(t1,...,tn) and P is non-fluent then

"(THNOT(THGOAL(P α(t1)...α(tn))(THTBF FILTERAX)))"

(iii) If l = P(t1,...,tn) and P is fluent then

"(THGOAL(P α(t1)...α(tn))(THTBF FILTEROP))"

where FILTEROP is a LISP function which controls the choice of rules or axioms entered on the basis of advice given, if any.

(iv) If l = ¬P(t1,...,tn) and P is fluent then

"(THNOT(THGOAL(P α(t1)...α(tn))(THTBF FILTEROP)))"

A Boolean expression of these statements corresponding to the precondition is generated. The implementation of conjunction and other functional parts of the theorem are described in later sections.

(11) The translated Frame is then loaded, i.e. functions, global variables and theorems defined and theorems asserted. The user may now begin program generation.

As an example of a translated rule consider the primitive procedure,

ROB(R1)∧¬KILLED(R1)∧AT(R1,L1)∧CLEAR(L1,L2)∨HASUMBRELLA(R1)
∧WALKABLE(L1,L2){walk(R1,L1,L2)}AT(R1,L2).

which translates into the Micro-Planner theorem shown in figure 19 which is labeled according to the basic functional segments described in Section 9.2.2 and shown in figure 18.

PATTERN

INTERACTIVE AND TRACE

NON-FLUENTS

ACHIEVEMENT OF FLUENT GOALS

UPDATING THE STATE

ASSERT P.C.

ASSEMBLE SOLN BLOCK STRUCTURE CHECKING

```
(DEFPROP WALK
         (THCONSE (CGL L1 L2 R1 D1 (LSTWALK (QUOTE (L2 R1))))
(AT (THV R1) (THV L2) R)
(THSETQ (THV LCTR) (THV GCTR))
(THUNIQUE LSTWALK)
(TREEPATH WALK (AT (THV R1) (THV L21 R))
(TRACEINFO1)
(THOR T (TRACEINFO2 WALK))
(COND ((TTYIN) (ADVICESYS)) (T T))
(THGOAL (ROB (THV R1)))
(THCOND ((THNOT (THGOAL (KILLED (THV R1) R))) T)
        (T (THGOAL (NKILLED (THV R1) R) (THTBF FILTEROP))))
(THCOND ((THAND (THASVAL (THV R1)))
         (THSETQ (THV NKILLEDARGS) (CONS (LIST (THV R1)) (THV NKILLEDARGS))))
        (T T))
(THGOAL (AT (THV R1) (THV L1) R) (THTBF FILTEROP))
(THCOND ((THAND (THASVAL (THV L1)) (THASVAL (THV R1)))
         (THSETQ (THV ATARGS) (CONS (LIST (THV R1) (THV L1)) (THV ATARGS))))
        (T T))
(THOR (THAND (THCOND ((THGOAL (CLEAR (THV L1) (THV L2) R) (THTBF FILTEROP)) T)
                     ((THGOAL (NCLEAR (THV L1) (THV L2) R)) (THFAIL))
                     (T
                      (UNCERTLIT (LIST (QUOTE CLEAR) (THV L1) (THV L2) (QUOTE R))1
                                 T
                                 (QUOTE (CLEAR (THV L1) (THV L2) R))
                                 (QUOTE (NCLEAR (THV L1) (THV L2) R)))))
             (THCOND ((THAND (THASVAL (THV L2)) (THASVAL (THV L1)))
                      (THSETQ (THV CLEARARGS)
                              (CONS (LIST (THV L1) (THV L2)) (THV CLEARARGS111)
                      (T T)))
      (THAND (THCOND ((THGOAL (HASUMBRELLA (THV R1) R) (THTBF FILTEROP)1 T)
                     ((THGOAL (NHASUMBRELLA (THV R1) R)) (THFAIL))
                     (T
                      (UNCERTLIT (LIST (QUOTE HASUMBRELLA) (THV R1) (QUOTE R))
                                 T
                                 (QUOTE (HASUMBRELLA (THV R1) R))
                                 (QUOTE (NHASUMBRELLA (THV R1) R))1)))
             (THCOND ((THAND (THASVAL (THV R1)))
                      (THSETQ (THV HASUMBRELLAARGS)
                              (CONS (LIST (THV R11) (THV HASUMBRELLAARGS))))
                      (T T)))
      (CONDSTAT (THV CGL) T))
(THCOND ((THGOAL (WALKABLE (THV L1) (THV L2) R) (THTBF FILTEROP1) T1
         ((THGOAL (NWALKABLE (THV L1) (THV L2) R)) (THFAIL))
         (T
          (UNCERTLIT (LIST (QUOTE WALKABLE) (THV L1) (THV L2) (QUOTE R))
                     NIL
                     (QUOTE (WALKABLE (THV L11 (THV L2) R))
                     (QUOTE (NWALKABLE (THV L1) (THV L2) R)))))
(CONDSTAT (THV CGL) NIL)
(THCOND ((THAND (THASVAL (THV L2)) (THASVAL (THV L1)))
         (THSETQ (THV WALKABLEARGS) (CONS (LIST (THV L1) (THV L2)) (THV WALKABLEARGS))))1)
        (T T))
(THCOND ((NULL (THV WALKABLEARGS)) T)
        (T (THSETQ (THV WALKABLEARGS) (CDR (THV WALKABLEARGS)) T T)))
(THCOND ((NULL (THV ATARGS)) T) (T (THSETQ (THV ATARGS) (CDR (THV ATARGS)) T T))1)
(THCOND ((NULL (THV NKILLEDARGS)) T)
        (T (THSETQ (THV NKILLEDARGS) (CDR (THV NKILLEDARGS)) T T)))
(THCOND ((THGOAL (HASUMBRELLA (THV R1) R))
         (THCOND ((NULL (THV HASUMBRELLAARGS)) T)
                 (T (THSETQ (THV HASUMBRELLAARGS) (CDR (THV HASUMBRELLAARGS)) T T1)))
        (T T))
(THCOND ((THGOAL (CLEAR (THV L1) (THV L2) R)1
         (THCOND ((NULL (THV CLEARARGS)) T)
                 (T (THSETQ (THV CLEARARGS) (CDR (THV CLEARARGS)) T T))))
        (T T))
(THCOND ((THAND (THASVAL (THV L2)) (THASVAL (THV L1))) T) (T (THFAIL11)
(THCOND ((EQUAL (THV L2) (THV L1)) (THFAIL)) (T T))
(THCOND ((THGOAL (AT (THV R1) (THV D1) R1) (THSETQ (THV ATINST) (LIST (THV R11 (THV D11111
        (T T))
(THCOND ((THGOAL (AT (THV R1) (THV D1) R)1 (THERASE (AT (THV R1) (THV D1) R) (THTBF THTRUE1))1
        (T T))
(THCOND ((THERASE (WRONG PATH)) (THFAIL THEOREM)) (T T))
(THSET (CAR (THV ANS))
       (CONS (CONS (QUOTE WALK) (LIST (THV R1) (THV L1) (THV L2))) (EVAL (CAR (THV ANS)11)))
(THSETQ (THV DBLITS) (CONS (CDAR CT) (THV DBLITS))1)
(THASSERT (AT (THV R1) (THV L2) R))
(THSETQ (THV ASSERTLITS)
        (CONS (LIST (LIST (QUOTE AT) (THV R1) (THV L2) (QUOTE R)) (LIST (QUOTE A) (QUOTE A)11
              (THV ASSERTLITS)))
(PRINT (REVERSE (EVAL (CAR (THV ANS)))))
(SETQ GANS (REVERSE (EVAL (CAR (THV ANS)))))
(COND ((>GREAT (LENGTH GANS) (LENGTH LGANS)) (SETQ LGANS GANS)) (T T)1
(THDO (TERPRI))
(COND ((EQ (QUOTE IF) (CADAR CT)1 (ELSECLAUSE)1 (T (THSETQ CT (CDR CT1 T T1111
THEOREM)
```

Figure 19

9.2.3.2 IMPLEMENTATION OF CONJUNCTION

The basic idea for implementing the achievement of a conjunction of goals, $G_1 \wedge G_2 \wedge \ldots \wedge G_n$, is to prevent the falsification of any G_i, $1 \leq i \leq n$, until all G_i are achieved, thus creating a state in which the conjunction is true.

For each fluent predicate symbol, say P, used there is a global variable created, i.e., PARGS, which is initialized to the value NIL and will hold a stack of instances of P that are to be preserved during the achievement of the conjunction. This is done by adding the instance(s) of the literal(s) whose achievement causes the current goal in the conjunction, say G_i , $1 \leq i < n$, to be true to the appropriate stack before G_{i+1} is attempted. When the entire conjunction has been achieved the literals for each G_i in that conjunction are popped from the stack. The LISP function that generates this code is recursive for arbitrary Boolean conditions satisfying the syntax.

A THERASING theorem is also generated for each P(X1,...,Xn) as follows:

```
(DEFPROP PGREMLIN
   (THERASING (X1...Xn)
      (P (THV X1)...(THV Xn)
      (THCOND((MEMBER(LIST(THV X1)...(THV Xn))
                  (THV PARGS))
            (THASSERT(WRONG PATH)))))
   (THEOREM),
```

where THV is a MICRO-PLANNER indicator that its argument is a variable.

If some instance P(t1,...,tn) is to be erased to maintain state consistency (see Section 9.2.3.4) then the act of erasing will call PGREMLIN which will assert the flag (WRONG PATH) into the data base if (t1,...,tn) is a member of PARGS. Such an assertion is responded to in the THCONSE theorem in which the erasure occurred by generating the following statement after the erase statement:

```
(THCOND((THERASE(WRONG PATH))(THFAIL THEOREM))(T T)).
```

The THERASE statement in the above will succeed only if (WRONG PATH) existed in the data base which was caused by an invalid erasure detected in the THERASING theorem. The flag seems necessary since success or failure in the THERASING theorem does not affect the success of the THCONSE theorem causing the erasure. The failure of the THCONSE theorem will force the system to try to find another theorem corresponding to another rule that does not falsify a goal in the conjunction.

9.2.3.3 CONTEXT LINKING

This feature discussed in Section 4 is implemented by denoting certain assertions in the data base as being "hypothetical" or not part of the state and used only in connection with this feature. If requested MICRO-PLANNER code is generated to precede the achievement of the precondition goals for rules and axioms and to carry out the following functions:

(1) The precondition goals are attempted relative to the hypothetical portion of the data base only to determine possible variable bindings.

(2) The instantiated precondition goals are asserted into the hypothetical data base for use in descendant rule applications in the subgoal tree.

Following the achievement of the preconditions of a rule, the hypothetical data base is restored to the state at rule entry.

9.2.3.4 UNIQUENESS PROPERTIES

Updating the state is discussed in Section 4 as an application of the invariance rule. "Building in" axioms defining uniqueness or single valuedness of certain relation argument position has proven useful for state consistency processing.

When a Frame is defined an argument position of any relation may be designated to be unique by responding to a system query with an asterisk in that position. Multiple argument positions may be so designated.

Before an instantiated postcondition, P(t1,...,ti,...,tn) is asserted, contradictory literals in the data base are removed. For each position designated as unique, suppose the ith, the goal P(t1,...,X,...,tn) is attempted with a new unbound variable in the ith position. If it is successful, i.e. X is now bound to val(X), then P(t1,...,val(X),...,tn) is erased.

For example consider the predicate AT(X,Y) = "Object X is at location Y", where both argument positions are unique, i.e. AT(*,*). Then in the state update portion of the theorem the following code is generated:

```
(THCOND((THGOAL(AT(THV X)(THV D1)))
        (THERASE(AT(THV X)(THV D1))(THTBF THTRUE)))
       (T T))
(THCOND((THERASE(WRONG PATH))(THFAIL THEOREM))
       (T T))
(THCOND((THGOAL(AT(THV D2)(THV Y)))
        (THERASE(AT(THV D2)(THV Y))(THTBF THTRUE)))
       (T T))
(THCOND((THERASE(WRONG PATH))(THFAIL THEOREM))
       (T T)),
```

where D1 and D2 are unbound variables.

This process assures that if the state is consistent with respect to uniqueness properties initially that this consistency will be maintained.

9.2.3.5 INTERNAL REPRESENTATION OF GENERATED PROGRAM

A program segment generated by the system is represented internally in a list data structure satisfying the following syntax:

```
<program>       ::= <block>
<block>         ::= (<statement-list>)
<statement-list> ::= <statement>|<statement><statement-list>
<statement>     ::= (<procname><arglist>)
<statement>     ::= (IF<condition> THEN <statement>)
<statement>     ::= (IF <condition> THEN <statement> ELSE <block>)
<statement>     ::= (WHILE <condition> DO <block>)
<statement>     ::= (← <identifier><expression>)
<procname>      ::= <identifier>
```

where,

<identifier> is an ALGOL identifier,
<expression> is a LISP functional expression in prefix form,
<condition> is a Boolean expression satisfying the syntax given in Section 3,
<arglist> is list of arguments each of which is either an <identifier> or an <expression>

For example,

((← X0 1)(← Y1 1)(WHILE ¬>(Y1 N) DO((← Y1 (ADD1 Y1))(TIMES X0 X0 Y1))),

is the factorial program in Section 7. The above syntax specification describes the structure of programs that may be generated by the system.

A partially generated program is actually maintained in a stack (a list with access only from the front) of "GENSYMed" variables which is pointed to by a global variable ANS. Each time a deeper level of nesting is required, i.e. to generate the body of a WHILE loop or nested conditional statements, a new variable name is added to the top of the stack and initialized to NIL. Program constructs generated at this level are assigned to the variable at the top of the stack via ANS using a THSET. When a level is emerged from the value of the top element is appended on to the value of the next-to-top element and the stack is popped.

When a generated program is output it is translated into a subset of ALGOL in the obvious way with nesting in the list structure corresponding to block nesting.

9.3 THE STATE UPDATING METHODS

The updating of a state to the new state resulting from the application of a frame rule is formulated by invariance which in general is not computable. Some of the more common causes of inconsistencies are handled by the uniqueness property mechanism described in Section 9.2.3.4. Also relevant to this topic is the discussion of conjunction implementation in Section 9.2.3.2. As explained in Section 6, updating the state after the application of an iterative rule may be either impossible or impractical unless the user provides an output assertion for the iterative rule in which case the rule of invariance is applied as with a primitive procedure postcondition. The results of applying the rule of invariance are influenced by the fixed, though arbitrary, ordering of the literals. To compute Inv(I,Q) a subset of I that is consistent with Q is sought. Since in general the choice of the $R_i \in I$ to be removed that prevents the derivation of a contradiction with Q is not unique, the ordering determines the deletion, if any.

The system philosophy has been that inconsistencies are of no concern unless they affect the correctness of the generated program. Consistent with this is a suggested approach that if an inconsistency is detected, say during some axiomatic deduction, that the choice of literals in I to be deleted be guided by the following,

(1) The information as to the state literals used to prove each previous goal as the program has been generated could be kept as an extension of the input-output assert computation (described later).

(2) The literals to be removed should be those that least affect the program, i.e. either those as yet unused or those most recently used since program generation would have to back up to that point then proceed after the deletion.

The actual Micro-Planner code generated to update the state after a primitive procedure has been applied is shown in figure 19.

9.4 COMPUTATION OF INPUT-OUTPUT ASSERTIONS

The computation of input-output assertions requires the extension of the MICRO-PLANNER system to include a trace stack containing rules entered, goals pending and goals achieved from the state, i.e. leaf nodes in the subgoal tree. This data structure is in addition to those which are a normal part of the MICRO-PLANNER Processor. This stack is a list data structure satisfying the following syntax:

<trace-stack> ::= (<rule-list>)
<rule-list> ::= <rule-use>|<rule-use><rule-list>
<rule-use> ::= ((<rule-name><current-goal>)<flag-list><achieved-goal-list>)
<achieved-goal-list> ::= <achieved-goal>|<achieved-goal><achieved-goal-list>

where,

<achieved-goal> is an instantiated precondition subgoal of the rule that has been achieved directly from the state,
<current-goal> is the current precondition subgoal pending in the rule for whose achievement rules above it in the stack have been entered,
<flag-list> is a sequence of zero or more flags used to determine proper block nesting in conditional statements (Section 9.5).

For example the trace stack may appear as,

(((T1 (P X1 a))(Q a))((T2 (R a X2))(S a b))),

at some stage of a computation and have the meaning,

(1) S(a,b) has been achieved from the state in T2,

(2) R(a,X2) is currently pending in T2 and T1 has been entered to attempt its achievement,

(3) Q(a) has been achieved from the state in T1, and

(4) P(X1,a) is currently being attempted.

As each rule, say T, is successfully applied, before its <rule-use> is popped from the trace stack, its <achieved-goal-list> is conditionally added onto a global variable.

DBLITS. Similarly if T has post conditions or output assertions to add to the state they are conditionally added onto a global variable, ASSERTLITS. The condition in both cases is that this occurrence of T will appear in the completed subgoal tree.

For any generated program segment A, the input assertion I_a and output assertion O_a may be computed as follows.

(1) By comparing each addition to DBLITS and ASSERTLITS in order of addition, those members of DBLITS that became true in the state as result of an assertion, (i.e. are members of ASSERTLITS), from a previous rule are deleted.

(2) Redundancies in DBLITS are removed yielding the input assertion I_a.

(3) The output assertion O_a is the non-redundant conjunction of all members of ASSERTLITS that are true with respect to the final output state of A.

9.5 GENERATION OF CONDITIONAL STATEMENTS

In Section 5 the algorithms for generating conditional statements were described. In this Section some of the details of the implementation will be given. Topics to be covered include implementation of goal nodes containing partial relations, contingency goal selection and its use, and association of rejoin conditions with contingency programs.

9.5.1 GOAL NODES CONTAINING PARTIAL RELATIONS

Let L be a precondition subgoal literal containing a partial relation. The code generated to attempt achievement of L is of the form:

```
(THCOND (α(L) T)
        (α(¬L) (THFAIL))
        (T (UNCERTLIT L SWITCH)))
```

where $\alpha(L)$ is the appropriate THGOAL statement from for L as described in Section 9.2.3.1; and UNCERTLIT is a LISP function or two arguments, i.e. an undetermined literal, L, and a switch value indicating whether this goal occurs in a conjunction (T) or in a disjunction (NIL).

The function UNCERTLIT does the following:

(1) Appends L to a global variable UNCERLIST,

(2) Returns not[SWITCH].

If L is in a disjunction then UNCERTLIT returns NIL, which forces the

next literal, if any, to be tried before the disjunction is declared undetermined and a conditional statement generated. See definitions in Section 5.1.

Either as the last statement of a THOR statement (which implements disjunction) or immediately following a THCOND statement like the above, a call to the LISP function CONDSTAT is generated with behavior:

(1) If null[UNCERTLIST] then if in a disjunction return NIL(causes failure) otherwise T(success).

(2) If not[null[UNCERTLIST]] then generate a conditional statement and contingency tasks as described in Section 5 and detailed in Section 9.5.2.

For example the disjunctive goal (see example in Appendix A),

VAR(x) V LP(x) V RP(x) V OP(x),

will result in the following code generated by the frame translator:

```
(THOR(THCOND((THGOAL(VAR(THV X)))T)
          ((THGOAL(NVAR(THV X))) (THFAIL))
          (T(UNCERTLIT(LIST(QUOTE VAR)(THV X))T)))
     (THCOND((THGOAL(LP(THVX)))T)
          ((THGOAL(NLP(THV X))) (THFAIL))
          (T(UNCERTLIT(LIST(QUOTE LP)(THV X))T)))
     (THCOND((THGOAL(RP(THV X))) T)
          ((THGOAL(NRP(THV X)))(THFAIL))
          (T(UNCERTLIT(LIST(QUOTE RP)(THV X))T)))
     (THCOND((THGOAL(OP(THV X)))T)
          ((THGOAL(NOP(THV X)))(THFAIL))
          (T(UNCERTLIST(LIST(QUOTE OP)(THV X))T)))
     (CONDSTAT(THV CGL)T))
```

where CGL is a variable having as value the post condition of the rule and is used in the contingency goal selection procedure. The goal

¬EMPTY(X),

occurring in a conjunction will result in the generation of the following code

```
(THCOND((THGOAL(NEMPTY(THV X)))T)
     ((THGOAL(EMPTY(THV X)))(THFAIL))
     (T(UNCERTLIT(LIST(QUOTE NEMPTY)(THV X))NIL)))
(CONDSTATE(THV CGL)NIL)
```

This code generated when the frame is translated will if executed at program generation time call the necessary construction procedures to generate conditional statements as further described in the next section.

9.5.2 IMPLEMENTATION OF CONDITIONAL STATEMENT GENERATION

When a goal is found to have an undetermined truth value as defined in Section 5 and implemented according to Section 9.5.1, the global variable UNCERTLIST is set to the list of undetermined literals (perhaps more than one literal if G is a disjunction). The following procedure is then carried out:

(1) The trace stack is searched from the top (current rule entered) to find the pending goal of smallest scope that is fully instantiated, say G* and to set (RPLACA) a flag,i.e. "IF", for each member of UNDERTLIST in the <rule-use> above G*, e.g. for rule names T and goals pending G,

(.. ((T G) IF ...) ((T' G*)...)...).

These flags in the trace stack will signal the end of the else clause and the point of rejoin for the contingency programs called from the conditional statement and generated later.

(2) The conditional statement is generated as described in Section 5.2. In particular for each member of UNCERTLIST, say L, a new procedure name, say p, is generated, the appropriate state, say S, is created and the triple (p,S,G*) is placed on the subproblem stack.

(3) An expression of the form,

(IF $\neg L_1$ THEN...(IF $\neg L_k$ THEN p_k ELSE p_{k-1})...ELSE)

is added to the top variable in the ANS stack. Note that the final else clause is left empty but will be filled in with what was called the trunk program segment in Section 5.2.

(4) The list of new procedure names generated in step (2) is "CONSed" to a global variable PROCLIST.

(5) A new answer variable name is generated added to the top of ANS and initialized.

(6) Program generation continues until a rule has been successfully applied that has some IF flags on its <rule-use> entry in the trace stack. The following steps are then carried out:

(a) Append the value of the top variable in ANS to the next-to-the-top variable and pop ANS. (note: This places the trunk program as the else clause)

(b) Form the triple ((CAR PROCLIST) DBLITS ASSERTLITS) using current values of these global variables and add it on to the variable PROCDATA to be used later to compute the rejoin condition for the programs named in (CAR PROCLIST)

REMARK: ANS and PROCLIST are managed as LIFO stacks which correspond to the entering and exiting of blocks in the generated program. This assures that the correct elements will always be at the top of the stacks and arbitrary depth of block nesting is allowed.

(7) After the trunk program has been completely generated, each triple in PROCDATA is accessed. Each consists of a list of procedure names having the same rejoin point in the trunk program, and the values of DBLITS and ASSERTLITS at the point of rejoin. The sufficient input assertion for the program segment from the point of rejoin to the may be computed by by removing from the values of DBLITS and ASSERTLITS at final output state their respective values at the point of rejoin then following the algorithm described in Section 9.4. This input assertion must be provable from the

final output state of each procedure in the list when it is generated (see Section 5.4) and is stored as an additional element of each associated triple in the subproblem stack.

9.6 ASSEMBLY OF WHILE LOOPS

In Section 8 problem reduction search in a THAND-OR-AND tree was described and in Section 6.1 the subgoal structure of a node expanded using an iterative rule was given, i.e. the premises that must be achieved to justify the construction of a loop. The subgoaling system provides program segments and substitution information allowing the loop assembly phase to fully satisfy the premises and construct a WHILE loop. This formal algorithm is sketched in Section 6 and now the methods implemented will be considered in more detail.

The inputs to the loop assembler will first be described. Next the system methods will be given for computing the successions of values for program variables to have during successive iterations of the loop. Here some of the methods are decidely neuristic in an effort to reduce the number of generated program variable and associated assignment statements. Then we describe the generation of the update assignment statements and their assembly with the other program segments to produce a complete while loop.

9.6.1 INPUTS TO LOOP ASSEMBLER

Consider an iterative rule applied to achieve I{?}G defined by the assertions P(basis), Q(invariant), R(iteration step goal), G(rule goal), L(control test) and S(output assertion) and where V is the list of variables in Q. The inputs required for loop assembly are as follows.

(1) A basis program segment p(P) is given that achieves the basis condition from state I, i.e. I{p(P)}I' and I' P.

(2) An instance Qλ of the loop invariant is given such that I'⊢Qλ, where $\lambda = \{\langle v_1 \leftarrow s_1 \rangle, \ldots, \langle v_n \leftarrow s_n \rangle\}$. The substitution λ is actually constructed by this

deduction and will be used to provide initial values for system generated program variables corresponding to certain v_i determined below.

(3) The formal algorithm calls for the generation of a loop body program segment p(R) that achieves the iteration step goal from the state $Q \wedge L$, i.e. $Q \wedge L\{p(R)\}I''$ and $I'' \subset R$. This is to assure that the generation of p(R) did not depend on particular properties of individual constants not shared by others of the same type in the domain and that p(R) would, in general, be incorrect. For example with respect to the integers zero has an additive property not shared by other integers, i.e. identity. In the current implementation $p(R\lambda)$ is generated such that $I'\{p(R\lambda)\}I'''$, where $Q\lambda \wedge L\lambda$ are true in I', then $p(R\lambda)$ is generalized as described below.

(4) An instance $Q\lambda'$ of the loop invariant is given such that $I'''\vdash Q\lambda'$, where $\lambda' = \{<v_1 \leftarrow t_1>,\ldots,<v_n \leftarrow t_n>\}$. Since the invariant is a characterization of relations in the subset of the state relevant to the iteration, comparing $Q\lambda$ and $Q\lambda'$ will reveal instances of value "changes" that should be computed using system generated loop control variables.

(5) An instance of the loop control test, i.e. $L\lambda$, is given.

In practice by taking the entire state I' as the input state from which to achieve R in step 3, the user's responsibility to express in Q all properties needed in the subgoal tree rooted at R is reduced.

9.6.2 COMPUTING SUCCESSIONS OF VALUES

It is assumed that the loop invariant Q characterizes the relations existing at each iteration among values of program variables. In particular all free variables in R and L must be among the free variables in Q. Therefore significant program variable value changes are given by comparing successive instances of Q, i.e. $Q\lambda$ and $Q\lambda'$ for the first iteration. If for each argument position in each relation in Q a different program variable is generated then a correct computation rule for updating the values in the program variables is a conditional assignment statement as described in Section 6 where each argument position in Q has a different w-variable. That some optimization (i.e. reduction of the number of program variables) could be done at program generation time is suggested by two observations:

(1) Many of the values in corresponding argument positions in $Q\lambda$ and $Q\lambda'$ will not change, i.e. they are constant for the loop,

(2) Many of those that do change may be controlled using the same program variable.

Since the frame language allows functional terms some successive values may be of the form, s_i goes to $f(s_i)$. In this case direct functional assignments of the form, $Y_i \leftarrow f(Y_i)$ may be efficiently placed at the top of the loop to avoid repeated computation. These ideas have led to a number of optimization heuristics which are intended to either:

(1) reduce the number of generated program variables,
or (2) recognize successive values related by a function and assemble direct functional assignments,
or (3) reduce the portion of Q required in a conditional assignment.

By comparing respective argument positions in $Q\lambda$ and $Q\lambda'$ the system recognizes two kinds of computation rules relating successive values, namely functional

computation, e.g. $t_i = f(s_i)$, and Boolean expressions, e.g. $T(s_i,t_i)$, where $T \subset Q$. The system constructs a list of significant change pairs each corresponding to one of the following cases:

(1) s_i and t_i are symbollic expressions related by the formula $T \subset Q$ and are represented by $((s_i, t_i)T)$.

(2) s_i and t_i are symbollic expressions related by a function f which is evaluated (using either EV or EVN), i.e. $t_i = f(s_i)$ and are represented by $(s_i, t_i, f(s_i))$. Note that in this case it is not sufficient to search terms in Q or R to find the function f. During the generation of p(R) the subgoal tree rooted at R is traced to return the function f, if any, used to compute a succession of values in the loop.

(3) s_i and t_i are symbollic expressions related by a function f which is not evaluated but left in symbolic form, and are represented by $(s_i, f(s_i), f(s_i))$.

9.6.3 ASSEMBLY OF PROGRAM SEGMENTS

Given the inputs specified in Section 9.6.1 and the change list described in Section 9.6.2, the loop assembly procedure does the following:

1. Generates a pair $<Y_i, Z_i>$ of control variables to take on the successions of values during loop execution for each change pair. This is to cover the case in which both s_i and t_i occur in p(R) and we want to avoid the complexity of considering statement order in p(R).
2. Constructs assignment statements that initialize the control variables prior to loop entry, their values for each execution of the loop so that an instance of the loop invariant will be true each time the loop body is entered,
3. Substitutes control variabbles for their values in the loop body
4. Assembles these proram segments together to form a "while" loop.

The detailed loop assembly procedure will now be given. The change pairs are given on a list CL and will either be of the form $((\alpha,\beta)T)$ or (α,β,F).

(1) Set PA to the first change pair on the change list CL. If all change pairs have been processed then go to (8).

(2) Generate a new pair of variables Y and Z to be used for predecessor and successor values respectively.

(3) Add $(Y \leftarrow \alpha)$ and $(Y \leftarrow Z)$ at the ends of p(P) and p(R) respectively.

Justification: The assignment $(Y \leftarrow \alpha)$ is an initialization of the variable Y to the initial value α and is done after the basis program p(P) prior to loop entry. The assignemtn $(Y \leftarrow Z)$ updates the variable Y after the iteration program IP with the successor of its former value which it is anticipated will be in Z in preparation for the next execution of the loop body.

(4) Add the replacement pair (α, Y) to the predecessor replacement list ALP and (β, Z) to the successor replacement list ALS for later substitution.

(5) If the change pair PA utilizes a function then add the assignment (Z ← F) to the successor function assignment list SASG, remove the first change pair from CL, and go to (1).

Justification: The function F is a fully instantiated function whose value is equivalent to β. This step causes Z to get the successor value as required in step (3).

(6) Generate a new variable W to be used as a call by reference variable in a conditional assignment statement and substitute W for all occurences of β in T.

Justification: W will hold the successor value for the conditional assignment to Z.

(7) Add the conditional assignment (IF T THEN Z ← W) to the conditional assignment list SASGR, remove the first change pair from CL and go to (1).

Justification: The relation T is assumed to specify the ordering between successive values that will be taken on by the control variables Y and Z, i.e. using T the successor of Y may be deduced. This of course implies the computability of T as a procedure call at execution time.

(8) Substitute variables for values in SASG and SASGR using the closure of ALP.

Justification: By closing the association lists under substitution dependence upon the order of substitution into SASG and SASGR is avoided. Subsitution into successor assignments only for predecessor values using their associated variables (Y's) is sufficient and in fact required because:

(a) Any successor value that may have occurred in a relation T has already been substituted for by W.

(b) A successor value is by our conventions the new value that is computed as a

result of executing the loop body and occurs as an argument in the invariant Q . By generating a distinct pair of control variables for each change pair, we separate the successor assignments so that each is a function of predecessor values only. Since the successor value of one change pair may be the predecessor of another this restriction is necessary.

(9) Substitute variables for values in p(R) and L using the closure of ALP annd ALS.

(10) Assemble a "while" loop in the following form:

```
p(P);
SASGR;
while ¬ L do
  begin
  SASG;
  p(R);
  SASGR;
  end
```

Remark: Ambiguities may arise because of equalities among elements in the change of values list, i.e. $((s_1, t_1) \ldots (s_k, t_k))$. There are thee cases, i.e.

(a) $\forall i,j\ [i \neq j \wedge s_i \neq s_j] \wedge \exists i,j[i \neq j \wedge t_i = t_j]$,

(b) $\forall i,j\ [i \neq j \wedge t_i \neq t_j] \wedge \exists i,j[i \neq j \wedge s_i = s_j]$,

(c) $\forall i,j\ [i \neq j \wedge s_i \neq s_j \wedge t_i \neq t_j] \wedge \exists i,j[i \neq j \wedge s_i = t_j]$.

These are resolved by referencing a trace of variable bindings in the subgoal tree associated with each occurence of each value or by simply re-achieving the iteration step condition R from state I" until the ambiguities disappear.

To illustrate the process of computing a succession of values generating successor assignments and substituting into them consider two examples from frames treated earlier.

Consider a slight variant of the iterative rule TUP in figure 12 and we have,

$Q\lambda$ = ON(M,B1,U) ∧ STACKED(B2,B1,U) ∧ SMALLER(B2,B1), and

$Q\lambda'$ = ON(M,B2,U) ∧ STACKED(B3,B2,U) ∧ SMALLER(B3,B2)

which results in a change pair of the form,

((B1,B2) STACKED(B2,B1,U) ∧ SMALLER(B2,B1)).

and the successor assignment, (after substitution using ALP)

```
IF STACKED(W1 Y1 U)∧SMALLER(W1 Y1) THEN
  Z1 ← W1;
```

As another example the iterative rule TFACT in figure 10 yields, (where we assume here that PROD is a primitive multiplication function)

$Q\lambda$ = C(X0,1) ∧ C(X1,0) ∧ FACT(1,0), and

$Q\lambda'$ = C(X0,(PROD 1(ADD1 0))) ∧ C(X1,(ADD1 0)) ∧ FACT ((PROD 1 (ADD1 0)), (ADD1 0)),

which results in the change pairs,

(0,(ADD1 0), (ADD1 0)) and (1,(PROD 1(ADD 0)),(PROD 1(ADD1 0)))

and successor assignments, (after substitution from closure of ALP and syntactic transformation from prefix functions as specified in the frame)

```
Y1 ← (Y1 + 1);
Y2 ← (Y2 * Y1);
```

After the loop has been assembled, control is given to an update procedure which applies the rule of invariance using the given output assertion S as previously described. If no output assertion is given then the loop is interpretively executed until the goal G is true. This is required to provide a correct initial state for continued program generation.

9.7 STRUCTURED PROGRAMMING

The objective of structured programming is to provide mental and organizational tools by which the programmer may create large systems while keeping the problem complexity firmly within his mental grasp at each step of the creation. In Section 7 the current rather rudimentary features in the system were briefly described and an example given.

Structured programming consists of constructing a program to solve a particular problem by specifying a sequence of operations in which the operations are not necessarily "primitive" to the interpreter, e.g. computer, human, etc., but if successfully carried our will correctly solve the problem. For each operation in the sequence that is not primitive i.e. the procedure is declared to be an assumption, the function it performs becomes a subproblem <p,I,G> for the system that may be similarly expanded into a sequence of perhaps again non-primitive operations. The process continues by step-wise refining each operation until the problem can be solved correctly using only "primitive" operations. The relationship between higher level operations and the equivalent sequences of sub-operations that may be generated by successive levels of structured development take the form of a tree with the initial generated program at the root.

During the structured development process an overall structure for the program is built up that primitive constructs will have to fit into. An implicit system assumption is that a lower level operation will not have side effects that affect the correctness of the overal structure containing it. This is essentially a "top-down" process, i.e., one proceeding from the general functional description level down to specification of primitives. However, there is a "bottom-up" component that occurs when on the basis

of information gained while generating lower level primitives, or to satisfy the requirements of using them, the overall structure, i.e., operations previously generated at a higher(closer to the root) level, must be modified. This may result in backtracking if these modifications invalidate any previously specified operation. Also the overall structure may be modified by shifting a high level operation specification to one which utilizes more mathematical properties of the problem domain. In the current system any bottom-up component and shifts(modifications) to higher level operations are done interactively by using the advice system. A useful automation of structured programming should provide more powerful control and record keeping facilities for the traversing the structured development tree.

The growing popularity of structured programming and its apparent usefullness for software understandability (and therefore reliability) indicates the need for continued research to automate this process. Certainly it is possible now to build an interactive structured programming system that can handle the top-down expansion, bottom-up backtracking and shifts at any level for the augmentation of the programmer.

APPENDIX A: EXAMPLES

1. A Simple Translator from Infix to Polish Notation

This example illustrates the generation of conditional branches within loops in a program to convert strings of symbols in infix form into strings in polish form, i.e. "(X+Y*Z)" converts to "XY+Z*". This is a common symbol manipulation task in a compiler. The example shows how the system can be used to program in a structured "top down" manner.

A fully parenthesized, syntactically correct infix expression of a specified length is given as input and on output a result stack S contains the Polish string. A working stack R is used during the translation. We may consider the basic data structures (stacks)i.e. variables, constructors,(e.g. push) and selectors (e.g. pop)),and the primitive operators as given. Then,in this case,the user proceeded in the following steps.

(1) First the actions of the top level of the program were described by declarative statements (i.e. the definitions of RECOGNIZED and PROCESSYM in terms of basic concepts such as "X is a left parenthesis", and intermediate concepts such as "pop operators from stack X and push them onto stack Y".

(2) Then at the second level, Rules - in this case iterative rules - were given for writing loops that implement the intermediate concepts. In doing this,the user specified the major characteristics of a loop and left the system with the details of deciding whether to write such a loop,and if so, with the choice of local variables,the actual operations in the loop body and their order,(in so far as that was not specified) and with looking after the updating of the local variables. Thus in order to write the top level loop, TSLOOP, to achieve POLTSL(T,U,V), the user must have "thought out" an invariant relation between the elements manipulated by the loop body and what the

goals of the loop body were (in this case one of the goals is a top level concept, RECOGNIZED(X,Y,Z)). The system, if it uses this rule in constructing the output, will construct a loop body including update assignments, and assemble it into a WHILE statement. Similary, in this example the user has supplied iterative rules for POPOPS and POPHOPS.

The output program consists of a main program, i.e. PROC1, containing a compound conditional statement which splits up the cases for processing as a function of the input symbol. Each allowable input symbol must be either of type variable, operator, left parenthesis, or right parenthesis. The main program proceses the case in which the input symbol is an operator and generates calls to contingency programs, PROC3, PROC4, & PROC5, to be generated for the other three alternatives. The procedure calls PROC2, PROC6, & PROC7 result in error exits.

The various parts of the Frame definition will be given below followed by the generated programs.

RELATIONS USED IN THE FRAME DEFINITION:

RELATION	INTERPRETATION	FLUENT	PARTIAL	UNIQUENESS
C(X,Y)	"Contents of X is Y"	TRUE	FALSE	C(X,*)
INTEGER(X)	"X is an integer"	TRUE	FALSE	FALSE
VAR(X)	"X is a variable"	FALSE	TRUE	FALSE
LP(X)	"X is a left paren"	FALSE	TRUE	FALSE
RP(X)	"X is a right paren"	FALSE	TRUE	FALSE
OP(X)	"X is an operator"	FALSE	TRUE	FALSE
ISVAR(X)	"X is a program variable"	FALSE	FALSE	FALSE
NEXTSYM(X)	"A value for X is input"	TRUE	FALSE	FALSE
RECOGNIZED(X,Y,Z)	"Symbol X is recognized wrt stacks Y & Z"	TRUE	FALSE	FALSE
PROCESSYM(X,Y,Z)	"Symbol X is processed wrt stacks Y & Z"	TRUE	FALSE	FALSE
>(X,Y)	"X is greater than Y"	FALSE	FALSE	FALSE
<(X,Y)	"X is less than Y"	FALSE	FALSE	FALSE
POLISH(X)	"X contains a Polish sequence"	TRUE	FALSE	FALSE
POLTSL(X,Y,Z)	"Translate an infix string x symbols long to Polish using stacks Y and Z"	TRUE	FALSE	FALSE
=(X,Y)	"X is equal to Y"	FALSE	FALSE	FALSE
PUSHED(X,Y)	"X is pushed onto Y"	TRUE	FALSE	FALSE
POPPED(X)	"X is popped"	TRUE	FALSE	FALSE
TOPPED(X,Y,Z)	"The top symbol of	TRUE	FALSE	TOPPED(X,Y,*)

	stack Y of size Z is assigned to X"			
POPOPS(X,Y)	"Pop operators from X and push onto Y"	TRUE	FALSE	FALSE
POPHOPS(X,Y,Z)	"Pop operators from Y that have greater priority than X and push onto Z"	TRUE	FALSE	FALSE
STACKSIZE(X,Y)	"Size of stack X is Y"	TRUE	FALSE	STACKSIZE(X,*)
STACK(X)	"X is a stack"	FALSE	FALSE	FALSE
EMPTY(X)	"Stack X is empty"	FALSE	TRUE	FALSE

ITERATIVE RULES:

NAME: TSLOOP
BASIS: NEWVAR(X,Y) ∧ C(X,0)
INVARIANT: C(X,W) ∧ INTEGER(W) ∧ STACK(V) ∧ STACK(U) ∧ ISVAR(Y)
ITERATION STEP: C(X,(ADD1 W)) ∧ NEXTSYM(Y) ∧ RECOGNIZED(Y,U,V)
CONTROL TEST: >(X,T)
OUTPUT ASSERTION: POLISH(V)
GOAL: POLTSL(T,U,V)

NAME: RLOOP
BASIS: NEWVAR(X) ∧ STACKSIZE(U,Z) ∧ TOPPED(X,U,Z)
INVARIANT: C(X,Y) ∧ =(Y,(TOP U)) ∧ STACK(U) ∧ STACK(V) ∧ STACKSIZE(U,W)
ITERATION STEP: PUSHED(X,V) ∧ POPPED(U) ∧ TOPPED(X,U,W)
CONTROL TEST: ¬OP(X)
OUTPUT ASSERTION: POPOPS(U,V)
GOAL: POPOPS(U,V)

NAME: OLOOP
BASIS: NEWVAR(X) ∧ STACKSIZE(U,T) ∧ TOPPED(X,U,T)
INVARIANT: C(X,Y) ∧ =(Y,(TOP U)) ∧ STACK(U) ∧ STACK(V) ∧ STACKSIZE(U,W)
ITERATION STEP: PUSHED(X,V) ∧ POPPED(U) ∧ TOPPED(X,U,W)
CONTROL TEST: ¬OP(X) ∨ <((PRIORITY X)(PRIORITY Z))
OUTPUT ASSERTION: POPHOPS(Z,U,V)
GOAL: POPHOPS(Z,U,V)

PRIMITIVE PROCEDURE	PRE-CONDITIONS	POST-CONDITIONS
push(X,Y) "Push symbol X onto stack Y"	ISVAR(X) ∧ STACK(Y) ∧ STACKSIZE(Y,Z)	PUSHED(X,Y) ∧ STACKSIZE(X,(SUB1 Y))
pop(X) "Pop stack X"	STACK(X) ∧ STACKSIZE(X,Y) ∧ ¬EMPTY(X)	POPPED(X) ∧ STACKSIZE(X,(SUB1 Y))
getnext(X) "Get next symbol"	ISVAR(X)	NEXTSYM(X)
←(X,Y) "Assign Y to X"	ISVAR(X)	C(X,Y)
top(X,Y) "Put top of stack Y in X"	ISVAR(X) ∧ STACK(Y) ∧ STACKSIZE(Y,Z)	TOPPED(X,Y,Z) ∧ C(X,(TOP Y))

DEFINITIONS:

BODY OF DEFINITION	RELATION DEFINED
(VAR(X) ∨ LP(X) ∨ RP(X) ∨ OP(X)) ∧ PROCESSYM(X,Y,Z)	RECOGNIZED(X,Y,Z)
VAR(X) ∧ PUSHED(X,Z)	PROCESSYM(X,Y,Z)
LP(X) ∧ PUSHED(X,Y)	PROCESSYM(X,Y,Z)
RP(X) ∧ POPOPS(Y,Z) ∧ POPPED(X)	PROCESSYM(X,Y,Z)
OP(X) ∧ POPHOPS(X,Y,Z) ∧ PUSHED(X,Y)	PROCESSYM(X,Y,Z)
=(X,0) ∨ INTEGER((SUB1 X))	INTEGER(X)

INITIAL STATE:

STACK(S) ∧ STACK(R) ∧ STACKSIZE(S,I) ∧ STACKSIZE(R,J)

ALGEBRAIC SIMPLIFICATION: (SUB1(ADD1 X)) → X

```
PROC1 (N R S)
ISVAR(X1);ISVAR(X2);ISVAR(X3);STACK(S);STACK(R);
COMMENT
INPUT:CONDITIONS:
STACKSIZE(R J)∧STACKSIZE(S I)
OUTPUT:CONDITIONS:
POLISH(S);
COMMENT
PROC6 ATTEMPTS:TO:ACHIEVE: (POPPED R)
PROC5 ATTEMPTS:TO:ACHIEVE: (PROCESSYM X2 R S)
PROC4 ATTEMPTS:TO:ACHIEVE: (PROCESSYM X2 R S)
PROC3 ATTEMPTS:TO:ACHIEVE: (PROCESSYM X2 R S)
PROC2 ATTEMPTS.TO:ACHIEVE: (PROCESSYM X2 R S) ;
  BEGIN
  X1 ← 0;
  WHILE ¬>(X1 N) DO
    BEGIN
    Z1 ← (X1+1);
    GETNEXT(X2);
    IF ¬OP(X2) THEN
      IF ¬RP(X2) THEN
        IF ¬VAR(X2) THEN
          IF ¬LP(X2) THEN
            PROC2(X2 R S)
          ELSE PROC3(X2 R S)
        ELSE PROC4(X2 R S)
      ELSE PROC5(X2 R S)
    ELSE
      BEGIN
      TOP(X3 R);
      WHILE OP(X3) ∧ ¬<((PRIORITY X3)(PRIORITYX2)) DO
        BEGIN
        PUSH(X3 S)
        IF EMPTY(R) THEN
          PROC6(R)
        ELSE
          BEGIN
          POP(R);
          END
        TOP(X3 R);
        END
      PUSH(X2 R);
      END
    X1 ← Z1
    END
  END

PROC3 (X2 R S)
ISVAR(X2);STACK(R);
COMMENT
```

```
INPUT:CONDITIONS:
STACKSIZE(R I)
OUTPUT:CONDITIONS:
STACKSIZE(R (ADD1 I))∧PUSHED(X2 R);
   BEGIN
   PUSH(X2 R);
   END

PROC4 (X2 R S)
ISVAR(X2);STACK(S);
COMMENT
INPUT:CONDITIONS:
STACKSIZE(S I)
OUTPUT:CONDITIONS:
STACKSIZE(S (ADD1 I))∧PUSHED(X2 S);
   BEGIN
   PUSH(X2 S);
   END

PROC5 (X2 R S)
ISVAR(X4);STACK(S);STACK(R);
COMMENT
INPUT:CONDITIONS:
STACKSIZE(R J)∧STACKSIZE(S I)
OUTPUT:CONDITIONS:
POPOPS(R S);
COMMENT
PROC7 ATTEMPTS:TO:ACHIEVE: (POPPED R) ;
   BEGIN
   TOP(X4 R);
   WHILE OP(X4) DO
      BEGIN
      PUSH(X4 S)
      IF EMPTY(R) THEN
         PROC7(R)
      ELSE
         BEGIN
         POP(R);
         END
      TOP(X4 R);
      END
   IF EMPTY(R) THEN
      PROC8(R)
   ELSE
      BEGIN
      POP(R);
      END
   END
```

2. Integer Square Root Problem

As an example of generating a program for numerical computation consider the task of computing the largest integer k for a given n such that k is less than or equal to the square root of n. An essential fact formalied in the Frame definition is that the difference between the itn and (i+1)st squares is 2i+1, i.e.

$$(I+1)^2 - i^2 = i^2 + 2i + 1 - i^2 = 2i + 1 = i + i + 1.$$

This allows the simple iterative upward computation for any i, using two variables Y1 and Y2 and only the arithmetic operation of addition, of i in Y1 and $(i+1)^2$ in Y2 such that when the value in Y2 exceeds n then Y1 will have the desired value k.

The Frame definition in addition to a primitive procedure for assignment is given below followed by the generated program.

RELATION	INTERPRETATION	FLUENT	PARTIAL	UNIQUENESS
C(X,Y)	"Contents of X is Y"	TRUE	FALSE	C(X,*)
>(X,Y)	"X is greater than Y"	FALSE	FALSE	FALSE
ISQRT(X,Y)	"X contains the integer square root of "Y"	TRUE	FALSE	ISQRT(X,*)
VSQ(X,Y)	"X equals Y "	TRUE	FALSE	FALSE
ISVAR(X)	"X is a variable"	FALSE	FALSE	FALSE

ITERATIVE RULE:

NAME:	TSQ
BASIS:	NEWVAR(X) ∧ C(X,(ADD1 0)) ∧ C(W,0)
INVARIANT:	C(W,Y) ∧ C(X,Z) ∧ VSQ(Z,(ADD1 Y))
ITERATION STEP:	C(W,(ADD1 Y)) ∧ C(X,(PLUS Z(ADD1(PLUS(ADD1 Y)(ADD1 Y)))))
CONTROL TEST:	≥(Z,V)
OUTPUT ASSERTION:	ISQRT(W,V)
GOAL:	ISQRT(W,V)

AXIOMS:

VSQ((ADD1 0),(ADD1 0))
VSQ((MINUS Z(ADD1(PLUS Y Y))),(SUB1 Y)) ⊂ VSQ(Z,Y)

INITIAL STATE:

ISAVR(X0)

ALGEBRAIC SIMPLIFICATION:

(SUB1(ADD1 X)) → X
(MINUS (PLUS X Y)Y) → X

```
PROC1(X0 N)
ISAVR(X0);
COMMENT
INPUT ASSERTION:
NONE
OUTPUT ASSERTION:
ISQRT(X0,N);
  BEGIN
  X0 ← 0;
  Y2 ← (0+1);
  WHILE ¬> (Y2,N) DO
    BEGIN
    X0 ← (X0 + 1);
    Y2 ← (Y2 + ((X0 + X0) + 1);
    END
  END
```

3. Hand-Eye Tasks

In a simple robotics environment an "eye" (usually a Vidicon TV camera) may be used to locate objects on a table and a computer contolled arm carries out manipulatory tasks with these objects. We assume the identity and location of the objects in the scene have been discovered and are given in the initial state.

Programs written for autonomous robot control must be capable of on carrying some sort of dialogue with the real world since most relations will be partial and the outcome of operations will not be totally reliable. Conditional calls to contingency procedures is one way of establishing this dialogue.

The frame definition is given below followed by a generated program.

RELATIONS USED IN THE FRAME DEFINITION:

RELATION	INTERPRETATION	FLUENT	PARTIAL	UNIQUENESS
AT(X,Y)	"X is at location Y"	TRUE	FALSE	AT(X,*)
HAS(X,Y)	"X has Y"	TRUE	FALSE	FALSE
CANREACH(X,Y,Z)	"X can reach from Y to Z"	TRUE	TRUE	FALSE
COLLIDED(X,Y,Z)	"X collided between Y and Z"	TRUE	FALSE	FALSE
DROPPED(W,X,Y,Z)	"W dropped X between Y and Z"	TRUE	FALSE	FALSE
AVAILABLE(X,Y)	"X is available at Y"	TRUE	TRUE	FALSE
MISSED(X,Y,Z)	"X missed Y at Z"	TRUE	FALSE	FALSE
ROBOT(X)	"X is a robot"	FALSE	FALSE	FALSE

PRIMITIVE PROCEDURE	PRE-CONDITIONS	POST-CONDITIONS
reach(A1,L1,L2) "A1 reaches from L1 to L2"	ROBOT(A1) ∧ OBJ(O1) ∧ HAS(A1,O1) ∧ ~ HAS(A1,O2) ∧ CANREACH(A1,L1,L2)	{AT(O1,L2) ∧ AT(A1,L2)}
transport(A1,O1,L1,L2) "A1 transports O1 from L1 to L2"	ROBOT(A1) ∧ OBJ(O1) ∧ HAS(A1,O1) ∧ AT(O1,L1) ∧ AT(A1,L1) ∧ CANREACH(A1,L1,L2)	{AT(O1,L2) ∧ AT(A1,L2)} ⊕ {COLLIDED(A1,L1,L2) ∧ DROPPED(A1,O1,L1,L2)}
pickup(A1,O1,L1) "A1 picks up O1 at L1"	ROBOT(A1) ∧ OBJ(O1) ∧ AT(O1,L1) ∧ ~ HAS(A1,O2) ∧ AVAILABLE(O1,L1) ∧ AT(A1,L1)	HAS(A1,O1) ⊕ MISSED(A1,O1,L1)
putdown(A1,O1,L1) "A1 puts down O1 at L1"	ROBOT(A1) ∧ HAS(A1,O1) ∧ AT(A1,L1)	~ HAS(A1,O1)

INITIAL STATE:

ROBOT(ARM) ∧ OBJ(BLK1) ∧ AT(BLK1,P) ∧ AT(ARM,B)

```
PROC1(BLK1 ARM P S)
ROBOT(ARM); OBJ(BLK1);
COMMENT
INPUT:CONDITIONS:
AT(BLK1 P) ∧ AVAILABLE(BLK1 P) ∧ CANREACH(ARM S P)
∧ CANREACH(ARM P S)
OUTPUT:CONDITIONS:
HAS(ARM BLK1) ∧ AT(ARM S) ∧ AT(BLK1 S);
   BEGIN
   IF ¬ AVAILABLE(BLK1 P) THEN
     PROC2(ARM BLK1)
   ELSE
     BEGIN
   IF ¬ CANREACH(ARM S P) THEN
     PROC3(ARM P)
   ELSE
     BEGIN
     REACH(ARM S P);
     IF ¬ AT(ARM P) THEN
       IF ¬ AT(ARM P) ∧ COLLIDED(ARM S P) THEN
         PROC4(ARM P)
       ELSE PROC5(ARM P)
     END
   PICKUP(ARM BLK1P)
   IF ¬ HAS(ARM BLK1) THEN
     IF ¬ HAS(ARM BLK1) ∧ MISSED(ARM BLK1 P) THEN
       PROC6(ARM BLK1)
     ELSE PROC7(ARM BLK1)
   END
IF ¬ CANREACH(ARM P S) THEN
   PROC10(BLK1 S)
ELSE
   BEGIN
   TRANSPORT(ARM BLK1 P S);
   IF ¬ AT(BLK1 S) THEN
     IF ¬ AT(BLK1 S) ∧ DROPPED(ARM BLK1 P S) THEN
       PROC11(BLK1 S)
     ELSE PROC12(BLK1 S)
   END
END
```

4. n-Queens Puzzle

To illustrate how the program generation system may be used to solve puzzles, a backtrack problem solving algorithm (see Section 9.1) is axiomatized in the frame definition language to solve the n-Queens puzzle. The object of this puzzle is to place n queens on an n x n chessboard such that they are mutually non-attacking. the algorithm proceeds by placing queens on the board a column at a time, backing up when no placement is possible.

The frame definition for this problem is given below followed by a generated solution programs for the 4-Queens and 8-Queens cases.

RELATIONS USED IN THE FRAME DEFINITION:

RELATION	INTERPRETATION	FLUENT	PARTIAL	UNIQUENESS
SAFE(X,Y)	"Square X,Y is safe"	TRUE	FALSE	FALSE
BOTHSAFE(W,X,Y,Z)	"Square W,X is safe wrt square Y,Z"	TRUE	FALSE	FALSE
ALLSAFE(X,Y,Z)	"Square X,Y is safe wrt columns 1,..,Z"	TRUE	FALSE	FALSE
QUEEN(X,Y)	"A queen is on Square X,Y"	TRUE	FALSE	FALSE
QPLACED(X,Y,Z)	"Queens are placed in columns X,..,Z"	TRUE	FALSE	FALSE
=(X,Y)	"X is equal to Y"	FALSE	FALSE	FALSE
PLACED(X)	"X queens have been placed"	TRUE	FALSE	FALSE

PRIMITIVE PROCEDURE	PRE-CONDITION	POST-CONDITION
placequeen(I,J) "Place queen on square I,J"	SAFE(I,J)	QUEEN(I,J)

AXIOMS:

ANTECEDENT	CONSEQUENT
=(J,1) ∨ {¬=(J,1) ∧ ALLSAFE(I,J,J)}	SAFE(I,J)
=(K,1) ∨ {REQUEST(QUEEN(IP,(EVN(SUB1 K)))) ∧ BOTHSAFE(I,J,IP<(EVN(SUB1 K)))∧ ALLSAFE(I,J,(EVN(SUB1 K)))}	ALLSAFE(I,J,K
¬=(I1,I2) ∧ ¬=((EVN(PLUS I1 J1)),(EVN(PLUS I2 J2))) ¬=((EVN(DIFFERENCE I1 J1)),(EVN(DIFFERENCE I2 J2)))	BOTHSAFE(I1,J1,I2,J2)

DEFINITIONS:

BODY OF DEFINITION	RELATION DEFINED
¬=(I,(EVN(ADD1 N))) ∧ ¬=(J,0) ∧ =(J,(EVN(ADD1 N))) ∨ {QUEEN(I,J) ∧ QPLACED(1,(EVN(ADD1 J)),N)} ∨ QPLACED((EVN(ADD1 I)),J,N)	QPLACED(I,J,N)
QPLACED(1,1,N)	PLACED(N)

INITIAL STATE: (empty)

```
PROC1
  BEGIN
  PLACEQUEEN(2 1);
  PLACEQUEEN(4 2);
  PLACEQUEEN(1 3);
  PLACEQUEEN(3 4);
  END


PROC1
  BEGIN
  PLACEQUEEN(2 1);
  PLACEQUEEN(5 2);
  PLACEQUEEN(7 3);
  PLACEQUEEN(1 4);
  PLACEQUEEN(3 5);
  PLACEQUEEN(8 6);
  PLACEQUEEN(6 7);
  PLACEQUEEN(4 8);
  END
```

APPENDIX B - AN INTERACTIVE SESSION

A sample interactive session is here presented to illustrate the system's use in frame definition and program generation. Statements typed by the user will always be prompted by "*". The top level system function is "SUBGOAL" which is called in the manner given below to accept a frame definition from the terminal. Comments to aid the reader's understanding of the dialogue will be enclosed in quotes.

```
*(SUBGOAL)
"The system now enters an interactive mode for Frame definition."
* * * * SEMANTIC FRAME DEFINITION * * * *

RULE TYPE* AXIOM
RULE NAME* AONTOP
IS THIS AN ASSUMPTION?* NIL
IS THE RULE DIRECTLY RECURSIVE?* NIL
INEQUALITIES IN ARGUMENT POSITIONS* NIL
PRECONDITIONS:
* ROBOT(X1) ∧ ON(X1,X2) ∧ ¬STACKED(X3,X2);
POSTCONDITIONS:
* ONTOP(X1);

RULE TYPE* PRIMITIVE PROCEDURE
RULE NAME* STANDON(R1,Z1)
IS THIS AN ASSUMPTION?* NIL
IS THE RULE DIRECTLY RECURSIVE?* NIL
INEQUALITIES IN ARGUMENT POSITIONS* NIL
PRECONDITIONS:
* ROBOT(R1) ∧ ¬ON(R1,W1) ∧ BOX(Z1) ∧ CLOTHES(O1) ∧ WEARING(R1,O1)
∧ AT(Z1,Y1) ∧ AT(R1,Y1);
POSTCONDITIONS:
* ON(R1,Z1);

RULE TYPE* PRIMITIVE PROCEDURE
RULE NAME* DRESS(R1,O1)
IS THIS AN ASSUMPTION?* T
IS THE RULE DIRECTLY RECURSIVE?* NIL
INEQUALITIES IN ARGUMENT POSITIONS* NIL
PRECONDITIONS:
* ROBOT(R1) ∧ CLOTHES(O1);
POSTCONDITIONS:
* WEARING(R1,O1);

RULE TYPE* PRIMITIVE PROCEDURE
RULE NAME* TRAVEL(R1,L1,L2)
IS THIS AN ASSUMPTION?* NIL
IS THE RULE DIRECTLY RECURSIVE?* NIL
```

INEQUALITIES IN ARGUMENT POSITIONS* (R1,*,*)
PRECONDITIONS:
* ROBOT(R1) ∧ AT(R1,L1) ∧ ¬ ON(R1,O2,L1);
POSTCONDITIONS:
* AT(R1,L2);

RULE TYPE* PRIMITIVE PROCEDURE
RULE NAME* STEPUP(X1,Y1,Z1)
IS THIS AN ASSUMPTION?* NIL
IS THE RULE DIRECTLY RECURSIVE?* NIL
INEQUALITIES IN ARGUMENT POSITIONS* (R1,*,*)
PRECONDITIONS:
* BOX(Z1) ∧ ROBOT(X1) ∧ STACKED(Z1,Y1) ∧ ON(X1,Y1);
POSTCONDITIONS:
* ON(X1,Z1);

RULE TYPE* ITERATIVE
RULE NAME* ITONTOP
IS THIS RULE DIRECTLY RECURSIVE?* NIL
BASIS CONDITION:
* ROBOT(X1) ∧ ON(X1,X2);
INVARIANT:
* ON(X1,X3) ∧ STACKED(X4,X3);
ITERATION STEP CONDITION:
* ON(X1,X4);
CONTROL TEST* NIL
OUTPUT ASSERTION* NIL
GOAL* ONTOP(X1);

RULE TYPE* NIL

INITIAL STATE:
* AT(M,CORNER) ∧ AT(B1,L) ∧ STACKED(B3,B2) ∧ STACKED(B2,B1)
∧ BOX(B3) ∧ BOX(B2) ∧ BOX(B4) ∧ STACKED(B4,B3) ∧ BOX(B1)
∧ ROBOT(M) ∧ CLOTHES(SHOES);

SEMANTIC PROPERTIES OF RELATIONS:

IS ROBOT(R1) A FUNCTION OF THE STATE?* NIL
IS ROBOT(R1) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* NIL

IS AT(R1,L1) A FUNCTION OF THE STATE?* T
IS AT(R1,L1) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* (R1,*)

IS STACKED(X4,X3) A FUNCTION OF THE STATE?* T
IS STACKED(X4,X3) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* (X4,*)

```
IS BOX(Z1) A FUNCTION OF THE STATE?* NIL
IS BOX(Z1) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* NIL

IS ONTOP(X1) A FUNCTION OF THE STATE?* T
IS ONTOP(X1) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* NIL

IS CLOTHES(O1) A FUNCTION OF THE STATE?* NIL
IS CLOTHES(O1) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* NIL

IS WEARING(R1,O1) A FUNCTION OF THE STATE?* T
IS WEARING(R1,O1) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* NIL

IS ON(X1,Z1) A FUNCTION OF THE STATE?* T
IS ON(X1,Z1) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* (X1,*)

FILENAME* DSK:PCLI
TRACE MODE?* T
PERFORMANCE STATISTICS?* T
LOOKAHEAD?* NIL
ALGEBRAIC SIMPLIFICATION?* NIL

SUBGOALING SYSTEM GENERATED!!!
"A subgoaling system corresponding to the Frame has now been generated
and the system may now receive a goal to achieve."

SUBMIT GOAL* ONTOP(M)
DO YOU WANT THE PROGRAM LIBRARY?* NIL
DO YOU HAVE ANY ADVICE?* T
*** ENTERING ADVICE SYSEM ***
#1* TRY STANDON BEFORE STEPUP
#2* NIL "Exit advice system and begin program generation."

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
---ITONTOP

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
---(ITONTOP(ON M X2))STANDON

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
---(ITONTOP(ON M X2))(STANDON(WEARING M SHOES))DRESS

((DRESS M SHOES))
"Current program segment generated is displayed in this form."

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
```

```
---(ITONTOP(ON M X2))(STANDON(AT M L))TRAVEL

((DRESS M SHOES)(TRAVEL M CORNER L))
((DRESS M SHOES)(TRAVEL M CORNER L)(STANDON M B1)
"This constitutes the basis program for the iterative rule."

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
---(ITONTOP(ON M B2))STANDON

STANDON IS FAILING!!!
---(¬ON M W1) WAS THE LOSER
"STANDON is only applicable for climbing from ground level."

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
---(ITONTOP(ON M B2))STEPUP

((STEP M B1 B2))
"This is part of the loop body."

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
---(ITONTOP(ONTOP M))AONTOP
"The system now interpretively updates the state until the goal is
true, then the while loop is assembled."

DO YOU WANT TO OPTIMIZE THE PROGRAM?* NIL
IS THIS PLAN USEFUL ENOUGH TO GENERALIZE?* T
IS THIS A PROCEDURE WITHOUT SIDE EFFECTS?* NIL

THE GOAL (ONTOP M) IS ATTAINABLE BY THE FOLLOWING PROGRM:
"The desired program has been generated and will now be displayed."

PROC1(M)
ROBOT(M);CL          SHOES);(BOX(B1);BOX(B2);
COMMENT
INPUT ASSERTIONS:
AT(M CORNER) ∧ AT(B1 L) ∧ STACKED(B2 B1)
OUTPUT ASSERTIONS:
WEARING(M SHOES) ∧ AT(M L) ∧ ONTOP(M);
COMMENT
THIS PROGRAM RELIES ON THE FOLLOWING ASSUMPTIONS:
(DRESS);
  BEGIN
  DRESS(M SHOES);
  TRAVEL(M CORNER L);
  STANDON(M B1);
  Y1 ← B1;
  IF STACKED(W1 Y1) THEN
    Z1 ← W1;
  WHILE ¬ ONTOP(M) DO
    BEGIN
```

```
    STEPUP(M Y1 Z1);
    Y1 ← Z1
    IF STACKED(W1 Y1) THEN
      Z1 ← W1;
    END
  END

DO YOU WANT TO DO STRUCTURED PROGRAM DEVELOPMENT?* T

TRYING---((DRESS M SHOES)(WEARING M SHOES)(STAT1.AST))
"This task triple consists of procedure name, goal and state."

DO YOU HAVE ANY ADVICE?* T

***ENTERING ADVICE SYSTEM***
#1* ADD PUT-ON

RULE TYPE* PRIMITIVE PROCEDURE
RULE NAME* PUT-ON(R1,O1)
IS THIS AN ASSUMPTION?* NIL
IS RULE DIRECTLY RECURSIVE?* NIL
INEQUALITIES IN ARGUMENT POSITIONS* NIL
PRECONDITIONS:
* ROBOT(R1) ∧ CLOTHES(O1) ∧ FOUND(R1,O1);
POSTCONDITIONS:
*WEARING(R1,O1);

RULE TYPE* PRIMITIVE PROCEDURE
RULE NAME* FIND(R1,O1,L1)
IS THIS AN ASSUMPTION?* NIL
IS RULE DIRECTLY RECURSIVE?* NIL
INEQUALITIES IN ARGUMENT POSITIONS* NIL
PRECONDITIONS:
* ROBOT(R1) ∧ CHAIR(O2) ∧ AT(O2,L1) ∧ AT(R1,L1) ∧ UNDER(O1,O2);
POSTCONDITIONS:
* FOUND(R1,O1);

RULE TYPE* NIL

INITIAL STATE:
* CHAIR(CHAIR1) ∧ CHAIR(CHAIR2) ∧ AT(CHAIR1,CORNER)
∧ AT(CHAIR2,CORNER);

SEMANTIC PROPERTIES OF RELATIONS:

IS FOUND(R1,O1) A FUNCTION OF THE STATE?* T
IS FOUND(R1,O1) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* NIL

IS CHAIR(O2) A FUNCTION OF THE STATE?* NIL
```

```
IS CHAIR(02) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* NIL

IS UNDER(01,02) A FUNCTION OF THE STATE?* T
IS UNDER(01,02) PARTIAL?* T
ARGUMENT UNIQUENESS PROPERTIES* NIL

ALGEBRAIC SIMPLIFICATION?* NIL

SUBGOALING SYSTEM GENERATED!!!
"The Frame addition has now been translated."

*2* DELETE DRESS
*3* NIL
"Exit Advice system."

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
---(PUT-ON(FOUND M SHOES))FIND

((FIND M SHOES CORNER))
((IF(¬UNDER SHOES CHAIR1) THEN (PROC2 M SHOES)
ELSE((FIND M SHOES CORNER)))(PUT-ON M SHOES))
"The conditional statement is generated since it is not known where
the shoes are."

DO YOU WANT TO OPTIMIZE THE PROGRAM?* NIL
IS THIS PROGRAM USEFUL ENOUGH TO GENERALIZE?* T
IS THIS PROCEDURE WITHOUT SIDE EFFECTS?* NIL

THE GOAL (WEARING M SHOES) IS ATTAINABLE BY THE FOLLOWING PROGRAM:
"This procedure is the structured expansion of the non-primitive
procedure DRESS called in PROC1."

DRESS(M SHOES)
ROBOT(M);CLOTHES(SHOES);CHAIR(CHAIR1);
COMMENT
INPUT ASSERTIONS:
AT(M CORNER) ∧ AT(CHAIR1 CORNER)
OUTPUT ASSERTIONS:
WEARING(M SHOES) ∧ FOUND(M SHOES) ∧ WEARING(M SHOES);
COMMENT
PROC2 ATTEMPTS TO ACHIEVE FOUND(M SHOES);
  BEGIN
  IF ¬UNDER(SHOES CHAIR1) THEN
    PROC2(M SHOES)
  ELSE
    BEGIN
    FIND(M SHOES CORNER);
    END
  PUT-ON(M SHOES)
```

```
  END

DO YOU WANT TO DO CONTINGENCY PLANNING?* T
WHAT IS YOUR PREFERENCE?
----IF NONE TYPE NIL* NIL

TRYING---(PROC2 (FOUND M SHOES)(STAT2.CST))
"The contingency task triple consists of procedure name, goal and state."

DO YOU HAVE ANY ADVICE?* NIL

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
---FIND
((FIND M SHOES CORNER))
DO YOU WANT TO OPTIMIZE THIS PROGRAM?* NIL
IS THIS PROGRAM USEFUL ENOUGH TO GENERALIZE?* T
IS THIS PROCEDURE WITHOUT SIDE EFFECTS?* NIL

THE GOAL FOUND(M SHOES) IS ATTAINABLE BY THE FOLLOWING PROGRAM:

PROC2(M SHOES)
ROBOT(M);CHAIR(CHAIR2);
COMMENT
INPUT ASSERTIONS:
AT(CHAIR2 CORNER) ∧ AT(M CORNER)
OUTPUT ASSERTIONS:
FOUND(M SHOES);
COMMENT
PROC3 ATTEMPTS TO ACHIEVE FOUND(M SHOES);
  BEGIN
  IF ¬UNDER(SHOES CHAIR2) THEN
    PROC3(M SHOES)
  ELSE
    BEGIN
    FIND(M SHOES CORNER);
    END
  END
DO YOU WANT TO DO CONTINGENCY PLANNING?* NIL
DO YOU WANT TO CONTINUE FROM THE CURRENT STATE?* NIL
```

REFERENCES

Allen, J., Luckham, D.C., "An Interactive Theorem-Proving Program," MACHINE INTELLIGENCE 5, B. Meltzer and D. Michie (Eds.), Edinburgh University Press, March, 1970.

Allen, J., Luckham, D.C., An unpublished working paper, AI Project Stanford University, 1973.

Balzer, R. . "Automatic Programming", Information Sciences Institute, Univ. Southern California, Technical Memorandum, September 1972.

Baumbart, B.G., "Micro-Planner Alternate Reference Manual". AI Project Operating Note, Stanford University, 1972.

Buchanan J.R., Luckham D.C., "On Automating the Construction of Programs", AI Project Memo, Stanford University, 1974.

Deutsch, P., Ph.D. Thesis, University of California at Berkeley, 1973.

Feldman, J. A., Low, J. R., Swinehart, D. C., Taylor, R. H., "Recent Developments in SAIL, An ALGOL Based Language for Artificial Intelligence, AI Memo AIM-176, Stanford University, 1972.

Fikes, R. E., Hart, P. E., Nilsson, N.J., "Some New Directions in Robot Problem Solving," MACHINE INTELLIGENCE 7, 1972.

Fikes, R. E., Nilsson, N. J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," ARTIFICIAL INTELLIGENCE, Vol. 2 (1971).

Floyd, R. W., "Assigning Meaning to Programs," Proc. of Symposium in Applied Mathematics, Vol 19, 1967.

Gerritsen, R., "Understanding Data Structures", Ph.D. Thesis, Carnegie-Mellon University, 1974.

Green, C. C., "Application of Theorem-Proving to Problem Solving," Proc. IJCAI, 1969.

Hammer, M.M., Howe, W.G., Wladawsky, I., "An Interactive Business Definition System", RC 4680, IBM Research, Yorktown Heights, NY, 1974.

Hewitt, C. , "Description and Theoretical Analysis of Planner" Ph.D. Thesis, M.I.T., 1971.

Hoare, C.A.R. , An axiomatic basis for computer programming, Comm. ACM, 12, 10, October 1969, 576-580, 583.

Hoare, C.A.R., and Wirth, N. , An axiomatic definition of the programming language Pascal, Berichte der Fachgruppe Computer-Wissenschaften 6, E.T.H., Zurich, November 1972.

Igarashi, S.; London, R.L.; Luckham, D.C. , "Automatic Program Verification I: A Logical Basis and Impiementation", Stanford AIM 200, May 1973.

Katz, S. M., Manna, Z., "A Heuristic Approach to Program Verification," Proc. IJCAi, 1973.

King, J., Floyd, R.W., " Interpretation Oriented Theorem Prover Over Integers", Second Annual ACM Symposium on Theory of Computing, 1970.

King, J., "A Program Verifier," Ph.D. Thesis, Carnegie-Mellon University, 1969.

Luckham, D.C., Buchanan, J.R., "Automatic Generation of Programs Containing Conditional Statements", AISB Summer Conference, Sussex, 1974.

Martin, W.A., Unpublished Working Paper, Project MAC, MIT, 1973.

McCarthy, J., and Hayes, P. , "Some Philosophical Problems from the Standpoint of Artificial Intelligence" Machine Intelligence 4, pp. 463-502, Edinburgh University Press.

Milner, R., "Logic for Computabie Functions Descriptions," AI Memo AIM-169, Stanford University, 1972.

Newell, A., Simon, H. A., "GPS, A Program that Simulates Human Thought," COMPUTERS AND THOUGHT, E. Feigenbaum and J. Feldman (Eds.), McGraw-Hill Book Co., 1963.

Nilsson, N., "Problem Solving Methods in Artficial Intelligence", McGraw-Hill, 1971.

Rulifson, J. A., Derkson, R. A., Waldinger, R. A., "QA4: A Procedural Calculus for Intuitive Reasoning," AI Center Tech. Note 73, SIR, 1972.

Samuel, A. "Studies in Machine Learning Using the Game of Checkers," COMPUTERS AND THOUGHT, E. Feigenbaum and J. Feldman (Eds.), McGraw-Hill Book Co., 1963.

Simon, H. A., "Experiments with a Heuristic Compiler," JACM 10 (Oct. 1963).

Stickel, M., "A Programmable Strategy Theorem Prover", Technical Report, Department of Computer Science, Carnegie-Mellon University, 1974.

Sussman, J., Winograd, T. , "Micro Planner Reference Manual", M.I.T. Project MAC Report 1972.

Sussman, G.J., Ph.D. Thesis, M.I.T., 1973.

Sussman, G. J. and McDermott, D. V., "Why Coniving is Better than Planning," Proc. FJCC 41 (Dec. 1972).

Tesler, L. G., Enea, H. J., Smith, D. C., "The LISP70 Pattern Matching System," Proc. IJCAI, 1973.

Waldinger, R. J. and Lee, R. C. T., "PROW: A Step Toward Automatic Program Writing," Proc. IJCAI, 1969.

Winograd, T., "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language," Tech. Report MAC TR-84, M.I.T., 1971.